SKROLLI 2014.2
SKROLLI.FI
Tietokonekulttuurin erikoislehti

Alkuun koodauksessa ja kolvauksessa

Omaa koodia Jollaan

Todellisuuden BINÄÄRIRAKENNE

Uusi juttusarja: Pelisuunnittelu

Muitakin oli kuin Kuusnepa

Sisällysluettelo

Ville-Matias Heikkilä
päätoimittaja

Puoli vuosisataa Basicia

Basic-kieli täytti tänä vuonna 50 vuotta, ja sen vaikutus tuntuu vahvana nykyäänkin.

Basicin taka-ajatuksena oli jo vuonna 1964 se, että ohjelmointi voisi olla helppoa ja välitöntä, muidenkin kuin erikoisasiantuntijoiden tajuttavissa olevaa. Vaikka monet ovatkin pitäneet Basicia sottaisena sormivärikielenä, sen vaikutusta kulttuuriin on turha kiistää.

Kun kotitietokoneita alkoi ilmestyä kauppoihin 1980-luvun vaihteessa, niiden puolestapuhujat vetosivat usein uuteen kansalaistaitoon, jopa uuteen lukutaitoon. Tietokoneita olisi kohta joka paikassa, ja koko kansan pitäisi oppia käyttämään niitä. Tähän käyttötaitoon kuuluisi toki erottamattomasti myös Basic-ohjelmoinnin hallitseminen.

Toisin kuitenkin kävi. Aikuiset eivät usein edes uskaltaneet koskea kotimikroihin. Lapset ja nuoret olivat rohkeampia, mutta lopulta vain pieni osa heistäkään rupesi kunnolla ohjelmoimaan. Ohjelmoimaan ryhtyneistä tuli uutta eliittiä, jonka oikkujen armoilla peruskäyttäjät joutuvat elämään. Basic oli edustanut hakkeri-idealistista unelmaa tasa-arvoisesta tietojenkäsittelystä, mutta se loikin perustan uudelle luokkajaolle.

Nyt, puoli vuosisataa Basicin jälkeen, ajatus ohjelmoinnista koko kansan yleissivistyksen osana on vihdoinkin tekemässä paluuta. Puhutaan esimerkiksi proseduraalisesta lukutaidosta, kyvystä lukea erilaisten algoritmisten rakennelmien toimintaperiaatteita. Maailmassa on proseduraalista ymmärrettävää yhä enemmän, ja tunnetusti paras tapa kehittää proseduraalista lukutaitoa on ohjelmointi. Ohjelmointia ei siis kannata opettaa kaikille pelkkänä satunnaisesti hyödyllisenä taitona, vaan siksi, että se antaa eväitä nykymaailman ymmärtämiseen.

Myös tässä Skrollin numerossa kunnioitetaan Basicin perintöä. Ohjelmointiasiaa on nimittäin nyt poikkeuksellisen paljon, ja ohjelmoinnin oppimisen teema toistuu useammassa jutussa. OpenGL-juttusarja päättyy, ja uusi pelisuunnitteluun keskittyvä sarja alkaa. Lisäksi mukana on ensimmäistä kertaa juliste – Oona Räisäsen maineikas esitys modeemin kättelyäänen rakenteesta. Mukavaa kesää lehden parissa ja muutenkin!

Skrolli
Tietokonekulttuurin erikoislehti

Yhteydenotot toimitus@skrolli.fi
Ircnet: #skrolli

Päätoimittaja Ville-Matias Heikkilä
Toimituspäällikkö Toni Kuokkanen
Toimitussihteeri Ninnu Koskenalho
Taiteellinen johtaja Risto Mäki-Petäys
Mediamyynti Jari Jaanto
Talous Anssi Kolehmainen

Muu toimitus Mitol Berschewsky, Tapio Berschewsky, Mikko Heinonen, Jukka O. Kauppinen, Ronja Koistinen, Sade Kondelin, Teemu Likonen, Annika Piiroinen, Kalle Viiri

Tämän numeron avustajat Kenneth Falck, Jyrki J. J. Kasvi, Sami Koistinen, Antti Laaksonen, Tapio Lehtimäki, Olli-Pekka Lehto, Asser Lähdemäki, Jussi Määttä, Olli Oikarinen, Sam Kh Pak, Annika Piiroinen, Harri Pitkänen, Manu Pärssinen, Ville Ranki, Mikko Rasa, Mikko Rauhala, Henna Ruohonen, Janne Sirén, Teija Tuhkio, Wallu, Antti J. Ylikoski

Julkaisija Skrolli ry

Painopaikka Tammerprint, Tampere,
ISSN 2323-8992 (painettu)
ISSN 2323-900X (verkkojulkaisu)

Kannen kuva:
Ville-Matias Heikkilä,
Risto Mäki-Petäys ja
Olli Oikarinen.

YMPÄRISTÖMERKKI
MILJÖMÄRKT
441 878
Painotuote

Laserleikkuri

– 3D-tulostimen skarppi, simppeli isoveli

Useimmille Skrollin lukijoille 3D-tulostin on tuttu laite. Jos sellaista ei ole itse päässyt käyttämään, niin niihin on päässyt tutustumaan vaikkapa Skrolli 2013.4:n artikkelissa tai internetin sivuilta. Toinen vastaava ja suosittu laite on laserleikkuri.

Teksti: Ville Ranki Kuvat: Ville Ranki, Mikko Niemikorpi

Laserleikkureita on käytetty teollisuudessa jo vuosikymmeniä, mutta Kiina-ilmiön ansiosta niiden hinnat ovat pudonneet harrastajien ulottuville. Laserleikkuri on yksinkertaisuutensa vuoksi myös helpompi rakentaa kuin 3D-tulostin. Skrolli kertoo kokemuksia laserleikkureista ja vertaa niitä 3D-tulostimeen. Verrokkilaitteina käytimme Vaasan Hacklabin pientä ja Tampereen Hacklabin isoa leikkuria.

Mikä laserleikkuri on?

Laserleikkuri on tietokoneella ohjattava, kahden akselin suuntaisesti liikkuva laseria ampuva laite. Käsittelemme artikkelissa erityisesti CNC-laserleikkureita, joissa CNC tarkoittaa ”computer numerical control”, tietokoneella ohjattavaa leikkuria. Vaasan leikkurissa laserputki pysyy paikallaan ja leikattavaa esinettä liikutetaan, kun taas Tampereen leikkurissa leikattava esine pysyy paikallaan ja säde ohjataan maaliinsa liikkuvien peilien kautta.

Laserilla pystyy leikkaamaan pehmeitä materiaaleja, kuten puuta ja muovia. Leikkaamisen lisäksi laserilla voidaan kaivertaa, eli piirtää materiaaleihin kuvioita, tehoa pienentämällä. Metalleja tämän teholuokan laserit eivät jaksa leikata, mutta esimerkiksi alumiiniin voidaan kaivertaa tekstiä.

Laserleikkuri on hyvin tarkka. Sen leikkauspäätä liikutellaan 3D-tulostimien tapaan askelmoottoreilla, joilla paikka pystytään asettamaan hyvin tarkasti. Säteen halkaisija on noin 0,1 mm, joten sillä pystyy tekemään hyvin tarkkoja viiltoja. Mittatarkkuus riittää esimerkiksi liimattomiin sormiliitoksiin, joissa osat pysyvät toisissaan kiinni pelkän kitkan avulla. Laserin leikkaamat esineet ovat teräväreunaisia ja sellaisenaan siistin näköisiä. Kaiverrusjälki esimerkiksi puussa on kaunista, ja laserilla onkin helppo tehdä erilaisia koriste-esineitä. Sen sijaan 3D-tulostettuja esineitä harvemmin voi kutsua kovin kauniiksi.

Käyttökohteita ja rajoituksia

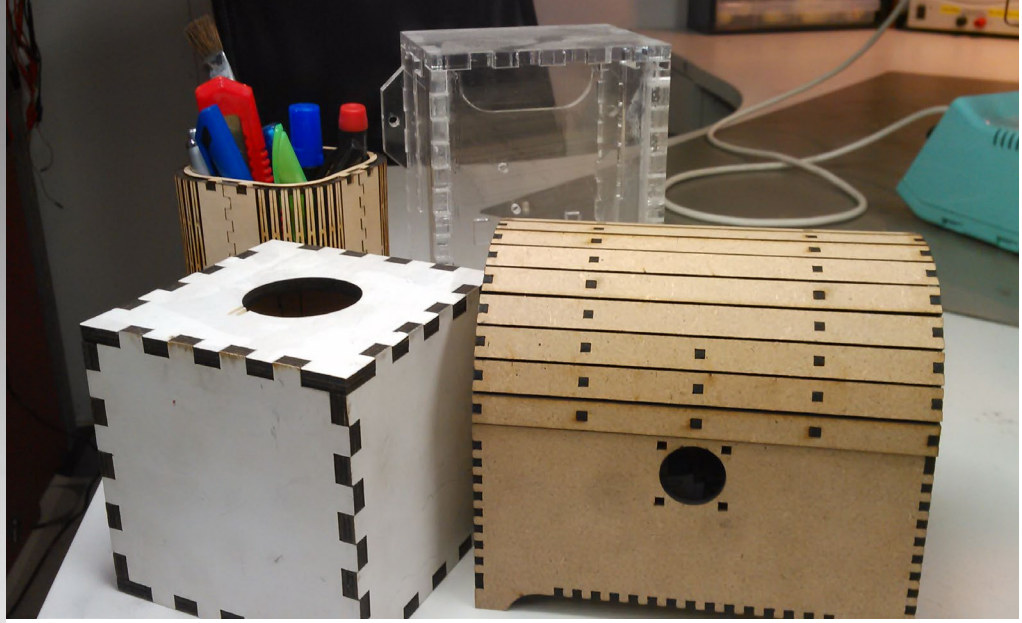
Laserleikkurin etuja ovat tarkkuus, nopeus ja laaja materiaalivalikoima. Hyvässä vireessäkään olevalla harrastajien 3D-tulostimella ei päästä mittatarkkaan tulostukseen, vaan osat vaativat usein viilausta sopiakseen täsmälleen yhteen. Laserleikkuri on myös huomattavan nopea, ja keskivertopalan leikkaus kestääkin vain minuutteja. 3D-tulostimella tulostusajat lasketaan kymmenissä minuuteissa tai tunneissa.

Käyttötarkoituksesta riippuen leikkurilla voidaan leikata esimerkiksi vaneria (lujaa), akryyliä (siistiä ja melko lujaa) tai lastulevyä (halpaa). Kokeiluleikkauksia- ja kaiverruksia voi tehdä vaikka roskiksesta kaivettuun aaltopahviin.

3D-tulostimeen verrattuna laserin suurin rajoitus on kaksiulotteisuus. Leikkuriin laitetaan yleensä tasapaksuista levyä, josta leikataan sopiva osa. Tätä voidaan kiertää kokoamalla kolmiulotteiset esineet kaksiulotteisista osista esimerkiksi sormiliitoksilla tai taivuttamalla. Tämä vaatii esineiden suunnitteluun hieman erilaista näkökulmaa. Esimerkiksi materiaalin paksuus vaikuttaa liitosten muotoon, joten se täytyy tietää etukäteen kappaletta suunnitellessa.

Laserleikkuri vaatii sijoituspaikaltaan vähän enemmän kuin 3D-tulostin. Leikkaamisesta aiheutuu hajua, joka on syytä poistaa ilmanvaihdolla. Hajuhaitan vähentämisen lisäksi leikkurin poistopuhallin imee savun pois leikkauspinnalta, jolloin se ei heikennä laserin tehoa. Kokemuksiemme mukaan puu haisee miellyttävimmältä, muovi pahalta ja nahka kaikkein vastenmielisimmältä. Laser siis täytyy sijoittaa tilaan, jossa on hyvä ilmanvaihto tai sellainen voidaan rakentaa vaikkapa viemällä ilmastointiputki ulos saakka.

Laserleikkurin sydän on näkymätöntä valoa tuottava hiilidioksidilaserputki, joka kuumenee käytössä merkittävästi. Laserin jäähdyttämiseksi tarvitaan jäähdytysvettä, joka pienemmissä malleissa



Laserilla tehtyjä kolmiulotteisia esineitä.

kierrätetään ämpäristä akvaariopumpulla ja isommissa suoraan vesijohtoverkosta. Veden unohtaminen saattaa aiheuttaa laserin ylikuumenemisen ja hajoamisen.

Tampereen iso laser vaatii lisäksi myös paineilmaa, jolla pidetään linssit ja peilit puhtaina. Tästä huolimatta ne ovat melko edullista kulutustavaraa.

Ei aivan vaaratonta

Laservalo on silmille vaarallista ja näkymätöntä. Tämän vuoksi leikkurit on yleensä koteloitu niin, että leikkauksen aikana ei ole vaaraa katsojille. Jos leikkuri ei ole koteloitu, täytyy samassa huoneessa olevien käyttää lasersuojalaseja. Vaasan leikkurissa tämä on ratkaistu rakentamalla kotelo leikkausalueen ympärille. Leikkauksen seuraamiseksi voi kotelon sisään asentaa kameran, jonka kuva näkyy leikkurin vieressä olevalta monitorilta.

Leikattavasta materiaalista täytyy tietää, soveltuuko se laserleikattavaksi. Suurimman vaaran aiheuttaa vinyyli eli PVC, jonka palamisesta syntyy kloorikaasua. Kloorikaasu syövyttää nopeasti leikkurin metalliosia ja on myrkyllistä hengitettynä. Tämän vuoksi esimerkiksi vanhoista LP-levyistä ei voi leikata koristeita. Useimmat materiaalit onneksi sopivat leikattaviksi, ja internetistä löytyy hyvin tietoa epäselvissä tilanteissa.

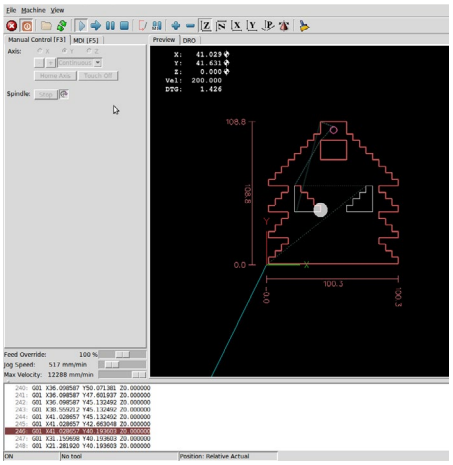
Oman projektin tekeminen

Koska laserleikkuri leikkaa kaksiulotteisia kappaleita, riittää kappaleiden suunnitteluun mikä tahansa 2D-vektoripiirto-ohjelma. Harrastajien suosiossa on avoin ja helppokäyttöinen Inkscape. Inkscape ei kuitenkaan ole täydellinen, sillä se on tarkoitettu taiteilijoiden, ei suunnittelijoiden työkaluksi. Esimerkiksi kappaleiden ääriviivat täytyy ottaa pois käytöstä, muuten niiden mitat lasketaan väärin. Hyvää ja helppoa Inkscapen korvaavaa CAD-ohjelmaa ei ole kuitenkaan vielä löydetty.

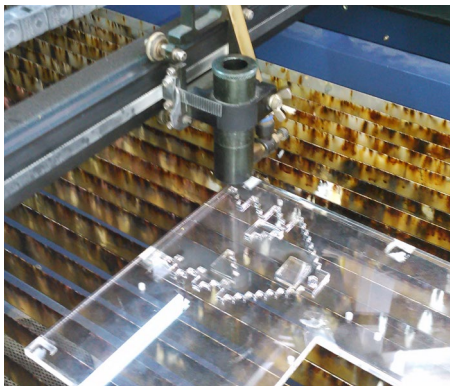
Laserleikkureiden ohjaus on hieman kirjavaa eikä niin tuotteistettua kuin 3D-tulostaminen. Pienempiä koneita, kuten Vaasan leikkuria, ohjataan HPGL-kielellä, joka on Postscriptin tapainen vektorikieli.

Leikkurille lähetetään HPGL-muotoon tallennettu mallinnustiedosto rinnakkaisportin kautta. Unixin kirjoitinpalvelu CUPS osaa muuntaa Postscriptiä HPGL:ksi, ja esimerkiksi Vaasan laserleikkuri näkyy tulostuspalvelimen avulla verkkotulostimena, jolle voi tulostaa suoraan kuten tavalliselle tulostimelle.

Jos leikkurin ohjaus osoittautuu toivottomaksi, on mahdollista vaihtaa sen ohjainkortti, kuten Tampereen leikkurin tapauksessa on tehty. Leikkurissa on Chris' Circuitsin valmistama kortti, joka



Skrollitrolli LinuxCNC-ohjelmassa valmiina leikattavaksi.



3D-tulostus oli vasta alkuvaiheessa, kun laserleikatut trollit olivat valmiina.

	Tampere	Vaasa
		
Malli	Yilong 1309B, modattu Chris' Circuitsin ohjainkortilla	Yilong S25
Teho	80 W	25 W
Leikkausala	130 × 90 cm	12 × 7 cm

Leikkurien tiedot.

tottelee 3D-tulostimien tapaan g-koodia. Tulostusta varten on kehitetty avoin LinuxCNC-jakelu, jota esimerkiksi Tampereen leikkuri käyttää. G-koodin generointiin käytetään Inkscapen lisäosaa, jolla luotu tiedosto avataan LinuxCNC:ssä leikkausta varten.

Laserleikkuria hankkiessa kannattaa varmistaa etukäteen, että sen käyttäminen on ylipäätään mahdollista halutuilla ohjelmistoilla. Leikkurin on tuettava haluttuja tiedostomuotoja tai ohjelmistoja, sillä laitteiden mukana toimitettavat kiinalaiset ohjelmat ovat yleensä toimivuudeltaan ala-arvoisia.

Valmiita laserleikattavia malleja löytyy esimerkiksi Thingiversestä (www.thingiverse.com). Halutunkokoisten sormiliitoksin kasattavien laatikoiden generointiin löytyy helppokäyttöisiä www-pohjaisia palveluja. Kannattaa kuitenkin muistaa, että materiaalin paksuus vaikuttaa kolmiulotteisten esineiden muotoon, mikä rajoittaa jonkin verran valmiiden mallien käytettävyyttä.

Kenelle laserleikkuri sopii?

Laserleikkuri vaatii vesijäähyineen ja ilmanvaihtoineen omistajaltaan selvästi enemmän vaivaa kuin 3D-tulostin, jota voi käyttää vaikka työpöydällä. Laserin ja 3D-tulostimen käyttökohteet ovat osittain samat, mutta laser voittaa selvästi materiaalivalikoimassa ja nopeudessa. Lisäksi 3D-tulostimella ei pysty kaivertamaan.

Laserleikkuri sopii parhaimmin hacklabeille ja tosiharrastajille, joilla on laitteelle käyttöä ja jotka voivat sijoittaa sen järkevästi. Jos omaa laseria ei pysty hankkimaan, kannattaa kysellä lähimmästä hacklabista tai turvautua internetistä löytyvien leikkauspalveluiden apuun.

Kitupiikin laserleikkuu

Risto Mäki-Petäys

Muiden hackerspacejen tavoin myös Helsinki Hacklabilla ilmeni kiinnostusta laserleikkaukseen. Jäsenet päättivät kokeilla edullisempaa ja ennakkoluulottomampaa ratkaisua ja tilasivat kiinalais-mongolialaisen laitteen.

Toivottavasti laitteen ohjelmisto ei edusta mongolialaisen ohjelmistosuunnittelun kärkeä. Käyttöliittymä on kokeneellekin käyttäjälle aika vaikeaselkoinen: täynnä pieniä nappeja, joissa on valmistusmaan kielellä olevaa tekstiä ja epäselviä pikkukuvia. Oli tuuria, jos sisältö näkyi edes suurimmassa osassa painikkeista. Kun kerran ohjelmistoa kokeillessani jokainen painike sattui näkymään oikein, se hetki ikuistettiin henkeä pidätellen kameralla ja kuvakaappauksella dokumentaatioon.

Kyseinen leikkuri on suunniteltu lähinnä erinäisten kumileimasinten tuotantoon, ja sen vuoksi laitteen mukana tuleva ohjelmisto suostuu käsittelemään vain yksibittisiä, mustavalkoisia, bmp-formaatin bittikarttakuvia. Tätä varten vektoroidun materiaalin siirto vaati useamman manuaalisen työvaiheen Inkscapesta Gimpin kautta leikkurin ohjelmistoon. Myöhemmin prosessia nopeutettiin skriptin avulla.

Laitteen turvallisuus on hieman kyseenalainen. Kannen ollessa kiinni siihen jää vielä suurehko oranssilla pleksillä peitetty aukko, jonka suojaavuudesta ei oltu aivan vakuuttuneita. Lisäksi heijastavasta materiaalista koostuvat palaset saattavat peilata sädettä rungon ulkopuolelle, kun ne tippuvat irti leikattavasta kappaleesta. Kaiken lisäksi valmistaja ei ollut nähnyt tarpeelliseksi lisätä interlock-kytkintä, joka estää laserin käynnistymisen kannen ollessa auki.

Pleksi päätettiin pinnoittaa kultakalvolla, joka suodattaa infrapunaa ja ultravioletia. Laitteen läheisyydessä työskenteleviä ohjeistettiin käyttämään suojalaseja. Myös laitteen muita rakoja hieman tilkittiin ja mainittu suojakytkin lisättiin laitteeseen. Suojakalvo ei tainnut olla täysin turha panostus, sillä se oli jo vähäisen käytön jälkeen hunnuttunut siksakkikuviolla.

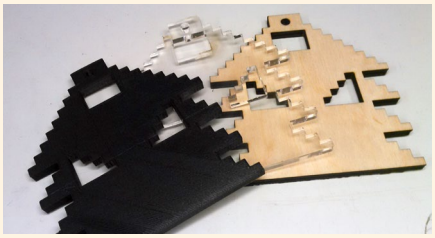
Näiden haasteiden jälkeen leikkuri toimi jonkin aikaa ilmeisesti kutakuinkin moitteetta, kunnes ohjelmisto päätti lopullisesti lakata toimimasta. Tämän johdosta hacklabin väki totesi, että leikkurin ohjaus lienee parempi korvata jollain järkevämmällä ratkaisulla.

Hyviä käytännön neuvoja laserleikattavista materiaaleista ja osista löytää Pololu-yhtiön ohjeesta osoitteesta http://www.pololu.com/docs/pdf/0J24/custom_laser_cutting.pdf.

Kilpailu: Laserleikkuri vs. 3D-tulostin

Teimme vertailun vuoksi samanlaiset esineet sekä 3D-tulostimella että laserleikkurilla. Samalla mittasimme eri työvaiheisiin kuluvan ajan. Tulostimena käytimme noin tuhannen euron hintaista, yleisesti käytettyä Ultimaker Originalia ja leikkurina Tampereen isoa laseria.

Tehtävä esine on noin 100×100×6 millimetrin kokoinen skrollitrolli, jota voi käyttää vaikkapa joulukuusen koristeena tai avaimenperänä. Vertailun lähtökohtana oli valmis svg- ja stl-tiedosto. Laitteet olivat valmiiksi päällä ja odottivat vain aloituskomentoa.



Skrollitrolli 3D-tulostettuna, akryylisenä ja vanerisena.

Testiä ei voi pitää mitenkään tieteellisenä tai reiluna, mutta se antaa jonkinlaisen käsityksen laitteiden nopeuseroista yhdessä esimerkkitapauksessa. Laserleikattu trolli tehtiin akryylistä ja tulostettava PLA-muovista. Tulostuksessa käytettiin tavanomaista 30 prosentin täyttöastetta ja 0,1 millimetrin kerrospaksuutta.

Tulokset eivät yllättäneet. Laserleikattu esine oli valmiina kädessä vain vähän myöhemmin kuin 3D-tulostin oli alkanut tekemään ensimmäistä kerrosta. Kaksi ja puoli tuntia kestäneen tulostuksen aikana olisi ehtinyt tekemään laseroimalla yli 30 trollia.

2010-luvun Commodore

Jyrki J. J. Kasvi
Tietoyhteiskunnan kehittämiskeskus TIEKEn tutkimus- ja kehitysjohtaja

Nokian sakattua moni on unohtanut, ettei Nokia ollut koko Suomi, ei edes Suomen koko it-sektori. Nokian varjossa Suomeen syntyi useita menestyviä it-alan yrityksiä etenkin tietoturva- ja pelialalle. Niihin ei vain kiinnitetty huomiota ennen kuin alettiin etsiä ”Uutta Nokiaa”.

Miten näin pieneen maahan on voinut syntyä niin monta it-yritystä ja Nokia vielä päälle? Kun tutustuu yritysten avainhenkilöiden historiaan, löytyy yksi yhdistävä tekijä: Commodore 64.

Tasavallan tietokone tuli markkinoille vuonna 1982. Sen ääressä lapsuutensa viettäneet ovat nyt noin 40-vuotiaita, ja monelle heistä Commodore 64 ja muut kahdeksanbittiset kotimikrot antoivat suunnan työuralle.

> *C64:llä pystyi tekemään oikeasti mielenkiintoisia asioita.*

Onneksi kaikki 80-luvun vanhemmat eivät noudattaneet ammattihuolestujien ohjeita ja repineet lapsiaan irti tietokoneesta sorkkaraudalla. Muuten olisi jäänyt moni työpaikka syntymättä.

Huolenlietsojien mielestä tietokoneiden kanssa vehtaaminen teki pojista joko epäsosiaalisia takahuoneen nörttejä tai psykoottisia tappokoneita. Tytöthän eivät moisiin härpäkkeisiin koskeneet. Moni tietokonetta vanhemmiltaan kärttänyt tyttö sai pikapassituksen ratsastusleirille, tytölle paremmin sopivan harrastuksen pariin.

Mikä Commodore 64:ssä sitten oli niin erityistä? Se oli juuri sopivan tehokas ja yksinkertainen. Sillä pystyi tekemään oikeasti mielenkiintoisia asioita, ja kuka tahansa pystyi ohjelmoimaan sitä, kunhan oli riittävän motivoitunut opettelemaan. Ja oltiinhan sitä.

Kynnys Basic-ohjelmointiin oli hyvin matala, ja kun sen oli oppinut, 6510:n konekielikään ei ollut ylivoimaisen vaikeaa.

Commodore 64:n sielunelämän tutkimista helpotti myös koneen suuri suosio Suomessa. Kaveripiiristä löytyi aina muitakin Kuusnepan omistajia, joiden kanssa saattoi vaihtaa kokemuksia, Mikrobittilehtiä ja opaskirjoja.

Yksi 1980-luvun suurimmista väärinkäsityksistä oli se, että kotitietokoneet tekisivät lapsista yksinäisiä ja epäsosiaalisia. Todellisuudessa kävi nimenomaan päinvastoin. Koneiden ääressä puuhattiin yhdessä.

Seuraukset näkyivät jo 1980- ja 90-lukujen demoskenessä. Demotiimit repivät tietokoneistaan kaikki tehot irti ja tekivät visuaalisesti vaikuttavia demoja, joiden ei olisi pitänyt olla teknisesti edes mahdollisia. Monet suomalaiset pelifirmat ovat saaneet alkunsa demoskenestä.

Mistä saisimme 2010-luvun lapsille oman Commodore 64:n? Mikä saisi tämän sukupolven samalla tavalla innostumaan tulevaisuuden tekniikasta ja ottamaan sen haltuunsa jo lapsena? Perustamaan yrityksiä ja luomaan työpaikkoja 2030-luvulla?

Jätämmekö vastuun taas kerran lapsille itselleen ja luotammeko, että he itse löytävät robotit ja 3d-tulostimet ja leikkivät niillä vanhempiensa vastustelusta huolimatta? Vai yritämmekö tällä kertaa aktiivisesti tuuppia lapsia ja nuoria, myös tyttöjä, tulevaisuuden suuntaan?

Kaikkia lapsia eivät tekniikka ja ohjelmointi kiinnosta. Heille on tarjolla musiikkiopistoja ja urheiluseuroja. Mutta miksi meillä ei ole samanlaisia koodikouluja niille lapsille ja nuorille, jotka eivät innostu ringetestä tai vetopasuunasta?

Lue lisää itse tekemisen riemusta ja 80-luvun kotitietokoneista tämän lehden artikkelista Tasavallan tietokoneen haastajat!

Älykkäämpää virheiden etsintää Voikolla

Tietokonepohjainen oikoluku on virtuaalinen punakynä, joka auttaa löytämään kirjoitusvirheitä. Perinteisesti ne tutkivat vain yksittäisiä sanoja, mutta tässä artikkelissa otamme askeleen eteenpäin.

Teksti: Harri Pitkänen Kuva: Mikko Rasa

Automaattinen oikoluku on jo pitkään kuulunut tekstinkäsittelyohjelmien perustoimintoihin. Sittemmin ominaisuus on yleistynyt myös muihin kirjoittamiseen käytettäviin ohjelmistoihin kuten nettiselaimiin ja sähköpostiohjelmiin. Hyvin toteutettu oikoluku onkin mainio apu tekstintarkistuksessa, ja siitä hyötyvät lähes kaikki kirjoittajat alakoululaisista ammattilaisiin.

Perinteisen oikoluvun toiminta-ajatus on yksinkertainen: Tekstissä olevat yksittäiset sanat analysoidaan sanaston ja muun logiikan avulla. Mikäli sananmuoto on oikolukujärjestelmälle tuttu, se katsotaan oikein kirjoitetuksi. Tuntemattomat sanat luokitellaan väärin kirjoitetuiksi ja merkitään tekstiin esimerkiksi punaisella alleviivauksella. Vuosien saatossa tämä perusidea ei ole juuri muuttunut.

Oikoluvun haasteita

Tekstin mekaaniseen tarkistukseen ja sanojen luokitteluun liittyy monia haasteita ja perustavanlaatuisiakin rajoituksia. Osa niistä on ratkaistu kauan sitten, mutta kaikkia ongelmia ei voi kiertää ohjelmallisesti.

Yksi vaikeus liittyy oikean ja väärän määritelmään, sillä ihmiselle ja koneelle ne ovat vähän eri asioita. Kone ei ymmärrä kirjoittajan ajatuksia, eikä tyypillinen oikoluku tutki tarkistettavan sanan asiayhteyttä. Sana vain hyväksytään, jos se löytyy ohjelman sanastosta. Käytännössä on kuitenkin mahdollista, että sana on muodollisesti oikein mutta asiayhteyden kannalta väärin. Tällaisen virheen voi yleensä huomata vain tekstin kokonaisuuden ymmärtävä ihminen.

Toisaalta oikolukuohjelmat tuomitsevat virheiksi myös oikein kirjoitettuja sanoja, koska ohjelma ei tunne kaikkia mahdollisia sanoja. Olisi kyllä mahdollista kartuttaa laaja sanavarasto, jonka avulla tekstin punaiset alleviivaukset katoavat. Niin ei kuitenkaan kannata tehdä, koska hyvin suuresta sanastosta on enemmän haittaa kuin hyötyä. Mitä enemmän sanoja ohjelma tunnistaa, sitä todennäköisemmin joidenkin sanojen taivutusmuodot tai yhdyssanatilanteet muistuttavat joitakin toisia sanoja. Tästä seuraa tilanteita, joissa lyöntivirheen seurauksena sanasta tulee jokin toinen sana, eli oikoluku ei havaitse kirjoittajan virhettä.

Oikolukuohjelman tekijälle erityisen haastavia ovat suomen kaltaiset kielet, joissa sanojen johtaminen ja taivutus on monipuolista ja joissa muodostetaan yhdyssanoja monin eri tavoin. Pelkästään näitä kieliopillisia ilmiöitä koskevan säännöstön kuvaaminen tietokoneohjelmalla voi olla erittäin työlästä.

Eikä se vielä riitä, että kerätään sanasto ja ohjelmoidaan sananmuodostusta säätelevä kielioppi. Oikolukuohjelman tekijä joutuu myös tutkimaan tekstejä, sanojen yleisyyttä ja tekemään monenlaisia valintoja. Välillä joudutaan punnitsemaan sanan tunnistamisen hyötyjä ja haittoja.

Esimerkiksi suomen kielessä sana *mangorove* on kieliopillisesti täysin oikein ja voisi tarkoittaa tuohesta valmistettua, mangohedelmien säilyttämiseen tarkoitettua rasiaa. Kuitenkin lähes varmasti sen kirjoittaja on tarkoittanut tropiikin rannikoilla esiintyvää mangrovekasvustoa mutta on kirjoittanut yhden *o*-kirjaimen liikaa. Koska jälkimmäinen vaihtoehto on todennäköisempi, oikolukuun ehkä kannattaa ohjelmoida poikkeus, jossa harvinaisen *rove*-sanan yhdyssanakäyttöä rajoitetaan. Samalla ehkä jotkin hyödylliset oikein kirjoitetut sanat tulevat hylätyksi. Kyse on kompromissien tekemisestä.

Tietokonepohjaiset oikoluvut eivät siis ole täysin luotettavia, ja laadukkaissa painojulkaisuissa tarvitaan edelleenkin huolellista ihmisen tekemää oikolukua ja kielenhuoltoa. Haasteet eivät tietenkään estä kehittämästä tekniikkaa paremmaksi.

Voikko

Avoimen lähdekoodin Voikko-oikolukuohjelmistoa on kehitetty aktiivisesti jo

Mikä on Voikko?

Voikko on vapaa oikolukuohjelma suomen kielelle. Se on käytännössä ohjelmakirjasto, joka tarjoaa oikoluvun, tavutuksen, morfologisen analyysin ja lauserakenteen tarkistustoimintoja. Muut ohjelmat voivat hyödyntää Voikko-kirjaston ominaisuuksia. Voikolle on olemassa ohjelmointirajapinta C-, C++-, Python-, Java-, C#- ja Common Lisp -kielille.

Kehittäjä: Harri Pitkänen
Lisenssi: GNU General Public License
Kotisivu: http://voikko.puimula.org/

kahdeksan vuotta, ja sitä voidaan pitää yhtenä laadukkaimmista suomen kielelle laadituista oikolukuohjelmista. Sitä levitetään muun muassa tunnetuimpien Linux-jakeluiden mukana, ja se on kytkettävissä moniin eri ohjelmiin.

Voikko ei kuitenkaan ole pelkkä oikolukuohjelma. Jos ohjelma tunnistaa tekstissä olevan sananmuodon, se pystyy myös kertomaan, mistä kantasanoista, johtimista, taivutuspäätteistä ja liitteistä sana muodostuu. Tätä kutsutaan morfologiseksi analyysiksi.

Tavanomaisissa tekstinkäsittelyohjelmissa käyttöliittymä ei mahdollista morfologisen lisätiedon hyödyntämistä mielekkäällä tavalla. Voikon perusversiosta onkin jätetty pois tavallisen oikoluvun kannalta merkityksetöntä lisätietoa, jotta ohjelmiston vaatima levytila ja muistin määrä pysyvät kohtuullisena. Ohjelmoijat voivat kuitenkin hyödyntää Voikon morfologista analyysiä ja kehittää sen avulla esimerkiksi vaihtoehtoisia oikolukumenetelmiä, jotka eivät rajoitu pelkästään sanan tulkitsemiseen kelvolliseksi tai virheelliseksi.

Älykkäämpi oikoluku

Tässä artikkelissa kehitämme esimerkkiohjelman, joka luokittelee sanat tavanomaista älykkäämmin. Sanat luokitellaan kahden sijasta kolmeen ryhmään: kelvolliset, epäilyttävät ja virheelliset. Esimerkkiohjelmassa hyödynnetään Voikon Python-ohjelmointirajapintaa.

Esimerkkiohjelman ajaminen onnistuu helpoimmin uusimmissa Linux-jakeluissa. Ensin on asennettava Voikon Python-tuki, joka esimerkiksi Debianissa ja Ubuntussa on paketissa python-libvoikko. Lisäksi tarvitaan morfologiseen analyysiin soveltuva morphoid-erityissanasto. Sen voi ladata sivulta http://www.puimula.org/htp/testing/voikko-snapshot/ ja asentaa purkamalla zip-paketti hakemistoon ~/.voikko. Ohjeita saa näkyviin Python-tulkissa komennolla import libvoikko; help(libvoikko).

Listauksessa 1 on esimerkki, kuinka Voikon oikoluvun perustoiminto eli sanan tarkistaminen toteutetaan Python-kielellä. Esimerkissä käytetään edellä mainittua morphoid-sanastoa. Listauksessa 2 esitellään Voikon analyze-metodia, jolla suoritetaan morfologinen analyysi. Ensin testataan sanaa *katuvalon*, josta saadaan yksikäsitteinen tulkinta: *katu* + *valo* + yksikön genetiivi eli omanto. Toisena analysoitavana on sana *valojen*, joka on monitulkintainen. Se voi olla taivutettu muoto sanoista *valo* tai *vala*.

Epäilyttävät sanat

Nyt kun perustoimintoja on kokeiltu, aletaan rakentaa tavallista älykkäämpää oikolukua. Ensin määritellään, mitkä sanat ovat epäilyttäviä. Olkoon niillä seuraavat ominaisuudet:

1. Perinteisen oikoluvun näkökulmasta sana on oikein.
2. Sana ei esiinny muualla tekstissä, ei edes muissa taivutusmuodoissa. (Mikäli sana esiintyy useasti, se ei todennäköisesti ole lyöntivirhe.)
3. Sanalle laskettu kompleksisuusarvo ylittää tietyn kynnysarvon. Mikäli sana on monitulkintainen, lasketaan kompleksisuusarvot kaikille tulkinnoille erikseen ja valitaan niistä pienin.

Ensimmäinen ehto on helppoa tarkistaa Voikon spell-metodilla. Toista ehtoa varten olisi mahdollista tarkistaa, ettei sanalla ole samaa perusmuotoa (eli BASEFORM-attribuutin arvoa, ks. listaus 2) kuin jollain toisella tekstissä esiintyvällä sanalla. Tällöin kuitenkin samaan kantasanaan perustuvat muodot (esimerkiksi *valo* ja *valoton*) tulkittaisiin kokonaan eri sanoiksi, mikä ei ole tämän tarkistuksen kannalta järkevää.

Lyöntivirheiden metsästyksessä erityisen kiinnostavia tapauksia ovat sanat, joissa kantasanakaan ei esiinny tekstissä muualla. Tämän selvittämiseksi voidaan hyödyntää analyysistä saatavaa WORDIDS-attribuuttia, joka sisältää viittaukset Voikon sanastotietokannassa oleviin tietuetunnisteisiin. Muodostetaan sanalle avainarvo näistä tunnisteista kirjoittamalla ne peräkkäin yhdeksi merkkijonoksi. Esimerkin sanalle *katuvalon* tämä avain olisi ”w504875w516884”. Sanalle *valojen* on kaksi mahdollista avainarvoa: ”w516832” ja ”w516884”.

Kolmatta ehtoa varten tarvitaan algoritmi kompleksisuuden laskemiseksi. Tämän voi tehdä monin eri tavoin. Käytetään nyt algoritmia, jossa kompleksisuus on verrannollinen sanan yhdysosien määrään ja kääntäen verrannollinen yhdysosien pituuteen. Tämä perustuu havaintoon siitä, että lyhyet sanat yhdys-

```
Python 2.7.6 (default, Feb 26 2014, 00:34:35)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or
"license" for more information.
>>> from libvoikko import Voikko, Token
>>> v = Voikko(u"fi-x-morphoid")
>>> v.spell(u"kissa")
True
>>> v.spell(u"kisssa")
False
```

Listaus 1. Oikolukua Python-kielen ja Voikon spell-metodin avulla.

```
>>> v.analyze(u"katuvalon")
[{u'BASEFORM': u'katuvalo', u'WORDIDS': u'+
katu(w504875)+valo(w516884)', u'SIJAMUOTO':
u'omanto', u'NUMBER': u'singular', u'CLASS':
u'nimisana', u'STRUCTURE': u'=pppp=ppppp',
u'WORDBASES': u'+katu(katu)+valo(valo)'}]

>>> v.analyze(u"valojen")
[{u'BASEFORM': u'vala', u'WORDIDS':
u'+vala(w516832)', u'SIJAMUOTO': u'omanto',
u'NUMBER': u'plural', u'CLASS': u'nimisana',
u'STRUCTURE': u'=ppppppp', u'WORDBASES':
u'+vala(vala)'}, {u'BASEFORM': u'valo',
u'WORDIDS': u'+valo(w516884)', u'SIJAMUOTO':
u'omanto', u'NUMBER': u'plural', u'CLASS':
u'nimisana', u'STRUCTURE': u'=ppppppp',
u'WORDBASES': u'+valo(valo)'}]
```

Listaus 2. Morfologinen analyysi sanoille **katuvalon** ja **valojen**.

```
def keyAndScoreForSingleAnalysis(analysis):
    if u"WORDIDS" in analysis:
        idParts = analysis[u"WORDIDS"].split(u"+")
        key = u""
        score = 1.0
        pattern = re.compile(u"^(.+)\((w[0-9]+)\)$")
        for idPart in idParts:
            match = pattern.match(idPart)
            if match:
                word = match.group(1)
                idString = match.group(2)
                key = key + idString
                score = score * (6.5 / len(word))
            elif idPart == u"-":
                score = score / 2.0
        return (key, score)
    return (u"", 0)
```

Listaus 3. Avainarvon ja kompleksisuuden laskeminen yhdelle analyysille.

```
def keyAndScore(analysisList):
    if len(analysisList) == 0:
        return (u"", 0)
    (key, score) = keyAndScoreForSingleAnalysis(analysisList[0])
    for analysis in analysisList[1:]:
        (newKey, newScore) = keyAndScoreForSingleAnalysis(analysis)
        if newKey == key:
            score = min(score, newScore)
        else:
            return (u"", 0)
    return (key, score)
```

Listaus 4. Avainarvon ja kompleksisuuden laskeminen useille analyyseille.

sanan osana muodostavat kummallisia kokonaisuuksia, jotka voivat olla merkki kirjoitusvirheestä. Lisäksi algoritmissa määritellään, että yhdysmerkki osien välissä pienentää kompleksisuuspistemäärää, koska yhdysmerkki on lyöntivirheenä harvinainen.

Koska sekä avainarvon että kompleksisuuden laskeminen perustuu sanan yhdysosien tutkimiseen, on ne käytännöllistä toteuttaa yhdessä funktiossa (listaus 3). Kaikkiin Voikon tunnistamiin sanoihin ei liity yhtään sanastotietokantaan tallennettua tietuetta (mm. pronominit), joten algoritmissa on varauduttu palauttamaan oletusarvona avaimeksi tyhjä merkkijono ja kompleksisuudeksi nolla.

Listauksen 3 funktio on sellaisenaan riittävä vain sanoille, joille Voikko tuottaa täsmälleen yhden morfologisen analyysin. Aiemmin kuitenkin nähtiin, että sanoilla voi olla useita analyysituloksia. Toisaalta kaikille sanoille (mm. väärin kirjoitetut sanat) ei analyysejä synny lainkaan. Listauksessa 4 esitellään funktio, jonka avulla nämäkin tapaukset saadaan käsiteltyä.

Näiden apufunktioiden varaan voidaan laatia aliohjelma, joka tulostaa annetusta tekstikappaleesta epäilyttävät sanat. Esimerkkikoodi on nähtävissä listauksessa 5. Sanat erotetaan juoksevasta tekstistä Voikon tokens-metodilla. Säätämällä kompleksisuuden enimmäisarvoa (tässä 2,7) tai muokkaamalla itse kompleksisuusfunktiota (listaus 3) voidaan vaikuttaa siihen, mitkä sanat tulkitaan epäilyttäviksi.

Jotta koodi olisi helpommin käytettävissä, kootaan funktiot samaan Python-tiedostoon, joka voidaan suorittaa käyttöjärjestelmän komentotulkista. Ohjelman runko on listauksessa 6. Kopioi sen sisään funktiot, jotka määriteltiin listauksissa 3–5.

Valmis esimerkki

Nyt on kasassa esimerkkiohjelma, jolla saa tulostettua tekstissä olevat epäilyttävät sanat. Esimerkkiohjelman voi ladata myös Skrollin nettisivulta osoitteesta http://skrolli.fi/2014.2/. Ohjelma poimii tekstistä sellaiset sanat, jotka ovat kieliopillisesti oikein mutta sen verran erikoisia, että kyseessä saattaa olla kirjoitusvirhe. Tekstistä löytyvät esimerkiksi kirjoitusvirheet *maksuase* (kun tarkoitettiin *maksutase*) ja *lumiura* (kun tarkoitettiin *lumiaura*). Ohjelma on kuitenkin vain yksinkertainen esimerkki. Sitä kokeilemalla huomaa, että monet sen merkitsemistä sanoista ovat aivan tavallisia suomen kielen sanoja.

Algoritmia voisi melko helposti parantaa ottamalla huomioon myös sanan osien tarkemman luokittelun. Esimerkiksi lyhenteet esiintyvät yhdyssanoissa yhdysmerkillä erotettuna, joten ne eivät käytännössä aiheuta hankalasti havaittavia kirjoitusvirheitä. Esimerkkialgoritmi ei huomioi sanojen luokittelua mitenkään, vaan kompleksisuuspisteytys kasvaa helposti suureksi, jos sanassa on mukana yhden tai kahden kirjaimen lyhenteitä. Jätämme tämänkaltaisen jatkokehittelyn haasteeksi lehden lukijoille. Vaihtoehtoisia algoritmeja voi lähettää Voikon kehittäjille, jos ne osoittautuvat toimiviksi.

```
def printSuspiciousWords(voikko, textParagraph):
    minScoreForSuspiciousWord = 2.7
    wordToKey = {}
    keyToScore = {}
    allTokens = voikko.tokens(textParagraph)
    words = [t.tokenText for t in allTokens if t.tokenType == Token.WORD]
    correctWords = [w for w in words if voikko.spell(w)]
    for word in correctWords:
        if word not in wordToKey:
            analysis = voikko.analyze(word)
            (key, score) = keyAndScore(analysis)
            wordToKey[word] = key
            if key in keyToScore:
                keyToScore[key] = 0
            else:
                keyToScore[key] = score
    for word in correctWords:
        if keyToScore[wordToKey[word]] >= minScoreForSuspiciousWord:
            print word.encode("UTF-8")
```

Listaus 5. Funktio, joka tulostaa epäilyttävät eli mahdollisesti väärin kirjoitetut sanat.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import fileinput, re
from libvoikko import Voikko, Token

# Kopioi tähän kohtaan listauksissa 3, 4 ja 5
# määritellyt funktiot.

voikko = Voikko(u"fi-x-morphoid")

for line in fileinput.input():
    printSuspiciousWords(voikko, unicode(line, "UTF-8"))

voikko.terminate()
```

Listaus 6. Pääohjelman runko, josta puuttuu listausten 3–5 koodi.

Voikko – suomalaisten NIH?

Teksti: Teemu Likonen

Avoimen lähdekoodin yhteisöissä on useamman kerran jouduttu selittämään ulkomaalaisille, miksi suomen kieli tarvitsee oman oikolukujärjestelmänsä. Miksi ette käytä Ispelliä? Onko teillä suomalaisilla NIH eli "not invented here" -oireyhtymä, jonka vuoksi muiden tekemä valmis koodi kelpaa? Ei. Meillä on vain tällainen vähän erilainen kieli...

Perinteisesti Unix-tyyppisissä käyttöjärjestelmissä oikoluku perustuu Ispell-järjestelmään, jonka juuret yltävät 1970-luvun alkuun. Ispelliin kuuluu kielioppi, jonka avulla sanasto ja sananmuodostus kuvataan. Unixin periaatteen mukaisesti siihen kuuluu myös komentotyökalu, joka toimii oikoluvun rajapintana muille ohjelmille.

Ja Ispellille todellakin on tehty suomen kielen sanasto. Samoin on tehty Aspellille ja Myspellille, jotka ovat eräällä tavalla Ispellin seuraajia. Miksi emme käytä niitä?

Laadun vuoksi. Käyttökelpoinen suomen kielen oikoluku vaatii hyvin taipuisan morfologisen järjestelmän, jotta suomen rikas taivutus ja sananmuodostus voidaan riittävän tarkasti määritellä. Ispell ja muut vastaavat ovat liian rajoittuneita. Tavallaan Voikko oli pakko tehdä, koska missään muualla ei kielestämme olla kiinnostuneita. Suurin kunnia ohjelmoinnista ja sanastotyöstä kuuluu Harri Pitkäselle ja Hannu Väisäselle.

Suomen kielen oikoluvun kannalta Voikko on ollut valmis jo pitkään, mutta sen tekniikalle on löytynyt muutakin käyttöä. Esimerkiksi pohjoissaamen kielelle on kehitteillä Voikko-sanasto. Kansalliskirjasto käyttää Voikkoa hakujärjestelmissään. Sen ansiosta palvelun käyttäjä voi kirjoittaa hakusanat perusmuodossa ja hakujärjestelmä löytää silti taivutettuja muotoja.

Avoimessa lähdekoodissa on etunsa, kun suomen kieliteknologian pyörää ei enää tarvitse jokaisen keksiä uudelleen.

Pelialan historiaa tutkitaan joukkorahoituksella

Kolme kirjaa avaa pelintekemisen todellisuutta.

Teksti: Jukka O. Kauppinen

Pelien tekeminenhän on helppoa. Naputtelee vain. Kohta on hitti käsissä ja eläkerahat pankkitilillä. Todellisuus on kuitenkin karumpaa, eikä kirjallisuudessa ole juuri avattu pelialan todellista historiaa, joka on värikästä, hauskaa ja joskus raadollistakin.

Joukkorahoituskampanjat ovat kuitenkin avanneet oven pelihistorioitsijoille aiheen syvällisempää tutkimusta varten. Sinivalkoinen pelikirja, Matkailua pelialalla ja Ocean The History ansaitsevat paikkansa pelihistoriasta kiinnostuneen kirjahyllyssä.

Sinivalkoinen pelikirja

Suomen pelialahistorian a–ö

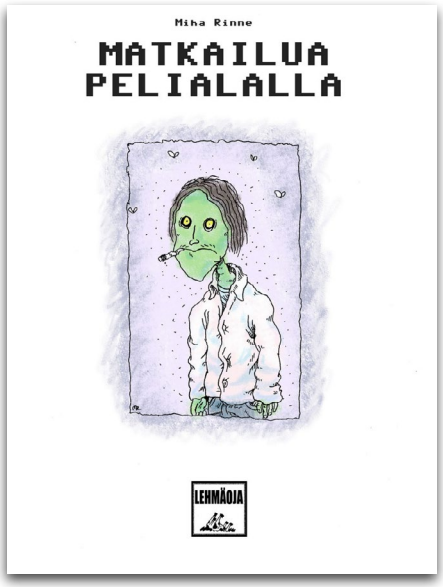
Juho Kuorikoski on tehnyt vuosien varrella hienoa pohjatyötä Sinivalkoisen pelikirjan runkoa varten. Kirjan julkaisemiseksi järjestettiin joukkorahoituskeräys, jonka vähimmäissummaksi asetettiin 2 500 euroa. Aihe selvästi kiinnostaa, sillä minimi ylittyi komeasti: kassaan kertyi yli 15 000 euroa.

Suomalaisen pelialan yli 30 vuotta kestänyt taival onkin nyt viimein käsitelty tavalla, joka kertoo, kuinka nykytilanteeseen on päästy. Arvostan Kuorikosken työssä erityisesti sitä, että heti alkuunsa kirja avaa kaupallisten suomalaispelien varhaisinta kirjoa ja muistuttaa, miten pienillä panoksilla ja pienille markkinoille ensimmäiset pelit tehtiin. Unohtamatta sitä, miten laajalti suomenkielisiä pelejä väsättiin – alustoja kun olivat sen tavanomaisen lisäksi myös VIC-20, MSX, Spectravideo ja ZX Spectum.

Digitaalisen esihistorian avaaminen on tehty taidolla ja kunnioittaen, eikä alan pimeämpääkään puolta ole pehmennelty. Unelma pelistä eli työttömyyskorvauksilla ja joskus koko tiimi eli toimistolla. Tai kellarissa. Unelma vei kuitenkin eteenpäin.

Lopulta unelma kasvoi aikuiseksi. Sinivalkoinen pelikirja kertoo tämän tarinan niin pelien kuin niiden tekijöidenkin kautta. Peliartikkelit, haastattelut ja eri aikakausien teemoja purkavat artikkelit rakentavat unelman tavoittelusta kasvukertomuksen, johon perehtymistä voin hartaasti suositella. Jotain tällaista olen itsekin miettinyt vuosia. Nyt minun ei enää tarvitse. Se on tehty.

Kirjailija: Juho Kuorikoski
Kustantaja: Fobos
Hinta: 43 euroa
Lisätietoja: http://www.fobos.fi



Matkailua pelialalla

Pelialan pimeä sarkastiikka

Miha Rinne on nähnyt suomalaisen pelialan synnyn – mutta onnekseen vai epäonnekseen, siinä kysymys. Rinteen viime vuonna joukkorahoitettu sarjakuva-albumi on näet mustaa huumoria pulppuava ja väkevän satiirinen näkemys pelialalla työskentelystä.

Satiiri kumpuaa Rinteen omista kokemuksista, jotka eivät ole olleet aina niitä kauneimpia. Unelma pelistä oli kaunis, mutta julkisivun takana oli ihmisiä, yhtiöitä ja kokemuksia, joista on pelottavaa miettiä, mikä pohjautuu toteen, mikä mielikuvitukseen.

”Jeff Minter taitaa olla ainoa oikeasti mukava ihminen koko kirjassa”, totesi puolisoni.

Pelialasarjakuva on ehdoton pari Sinivalkoiselle. Ne kertovat saman tarinan kaksi eri puolta, toinen dokumentaarisemmin, toinen sarjakuvadraaman kautta.

Kirjailija: Miha Rinne
Kustantaja: Lehmäoja
Hinta: 35 euroa
Lisätietoja: http://peliala.wordpress.com

Ocean The History

Ocean oli kuin eurooppalainen kellari-EA

Brittiläinen Ocean Software oli aikoinaan Euroopan suurin ohjelmistotalo. Keskellä kotimikrobuumia perustettu pelitalo oli aikansa kapinallinen, josta kasvoi legenda. Ocean operoi Britannian pelialan reunamilla Manchesterissa ja tunki pelintekijänsä tupakansavuiseen kellariin, The Dungeoniin, jossa syntyivät niin Euroopan parhaat kuin kamalimmatkin videopelit.

Moni muistaa Oceanin ääripäistään: Robocop, The Great Escape, Head Over Heels ja Rambo First Blood Part II olivat supermenestyksiä. Knight Rider ja Street Hawk hirveyksiä. Kaikkeen on kuitenkin selityksensä, ja kahden brittiläisen pitkän linjan pelitoimittajan Kickstarterissa rahoittama teos avaa Oceanin historiasta saloja, joita minäkään en ole kyennyt edes kuvittelemaan.

Upeasti toimitettuun kirjaan on haastateltu 40:tä Oceanin työntekijää, joiden muistelot, detaljit ja valokuvat avaavat rakastetun yhtiön tarua ainutlaatuisella tavalla. Ocean oli pelitalo, jossa sattui ja tapahtui – ja heilläkin oli unelma pelistä.

Kirjailija: Chris Wilkins & Roger M. Kean
Kustantaja: Revival Retro Events
Hinta: 5,99 puntaa (pdf), 25 puntaa (kirja)
Lisätietoja: http://www.oceanthehistory.co.uk

Ohjelmoinnin opintiet

Moni haluaisi oppia ohjelmoimaan. Oppimistavoista on kuitenkin monenlaisia näkemyksiä.

Teksti: Ville-Matias Heikkilä

Kuvat: Marko Vallius, Susan Pogany, Wikimedia Commons -käyttäjä Tanderson11

Tilanne on varmasti monelle tuttu. Ohjelmoinnin oppimisesta kiinnostunut kohtaa kokeneen ohjelmoijan ja pyytää tältä neuvoa alkuun pääsemiseksi. Kokenut ohjelmoija muistelee omaa alkutaivaltaan ja neuvoo aloittelijan samankaltaiselle polulle. Kumpikaan ei ymmärrä, että ohjelmoinnin oppimiseen on monia eri tapoja ja yhden tapa saattaa olla toiselle täysin mahdoton.

Tämän artikkelin tarkoitus on esitellä muutamia mahdollisia reittejä ohjelmoinnin oppimiseen etenkin omatoimisen opiskelun näkökulmasta. Millaisella kielellä kannattaa aloittaa, millaisella filosofialla jatkaa siitä eteenpäin ja millaisessa sosiaalisessa ympäristössä? Vastaukset näihin kysymyksiin ovat hyvin yksilöllisiä.

Lelukielellä alkuun

Ohjelmointikielet ovat usein vaikeita ja pitkäpiimäisiä. Yksinkertaisenkin asian tekeminen voi vaatia sellaisen määrän ympäryskoodia, että aloittelija hätääntyy täysin. Teorian, syntaksin ja rajapintojen kiemurat saavat usein opiskelussa pääosan, ja itse ohjelmointi hukkuu niiden alle. Tämän vuoksi ohjelmointiin voi olla hyvä hakea ensin tuntumaa jonkinlaisella lelukielellä.

Yleinen harhakäsitys on, että ohjelmoinnin osaaminen olisi sama asia kuin jonkin ohjelmointikielen osaaminen. Ohjelmointitaidon perusta on kuitenkin kielistä riippumaton, eräänlainen niksahdus aivoissa. Kun tämä niksahdus on jonkin kielen avustuksella saatu aikaan, ei sitä tarvitse enää tehdä uudestaan myöhempien kielten kohdalla. Tästäkin syystä on hyvä keskittyä aluksi helppoon lelukieleen.

Etenkin nuorimmille oppijoille on luontevaa suhtautua lelukieliin kuin mihin tahansa leluihin: vähät väliä aikuisten teorioista ja ammattikäytännöistä, tärkeintä on päästä kokeilemaan ja tekemään itse. Jälki voi etenkin aluksi olla sottaista, mutta ymmärrys syvenee ja käsiala paranee kokemuksen myötä – ainakin toivottavasti. Harhapoluille eksytään usein, mutta se ei haittaa, kun innostusta riittää. Hyvän lelukielen oleellinen piirre onkin innostuksen ylläpito esimerkiksi helpon grafiikkaohjelmoinnin kautta. Teoreettisempien ja vähemmän leikkimielisten oppijoiden kannattaa kuitenkin suhtautua lelukieliinkin systemaattisemmin.

Ehkä tunnetuin lelukieli on vuonna 1964 Dartmouthin yliopistossa alkunsa saanut Basic (Beginner's All-purpose Symbolic Instruction Code). Kieli syntyi eräajojen ja reikäkorttien aikana, jolloin harvat pääsivät suoraan vuorovaikutukseen tietokoneen kanssa. Basicin perusajatuksena oli kuitenkin alusta asti vuorovaikutteisuus: opiskelijat istutettiin päätteiden ääreen, ja tietokone vastasi käskyihin ja koodimuutoksiin välittömästi.

Kun ensimmäisiä mikrotietokoneita ruvettiin rakentamaan 1970-luvulla, olivat rakentajat usein saaneet ensikosketuksen ohjelmointiin nimenomaan Basicin kautta. Ei siis ihme, että sisäänrakennetusta Basic-tulkista tuli kotitietokoneissa itsestäänselvyys – niin hyvässä kuin pahassa.

Monien Basic-suvun kielten ympärillä on edelleen aktiivisia yhteisöjä. Etenkin peliohjelmointi on aina ollut suosittua Basic-harrastajien keskuudessa. Peli- ja grafiikkakeskeiseen aloitteluun sopivat basicien lisäksi esimerkiksi lapsille tarkoitetut Scratch ja Etoys, mediataiteilijoiden suosima Processing ja pelintekoohjelma Game Maker GML-kielineen.

Kotisivujen teko HTML-kielellä on toiminut monille porttina ohjelmointiin, vaikkei pelkkä HTML vielä ohjelmointikielen määritelmää täytäkään. Www-selain on nykyisin kaikenlaisten koneiden vakiovaruste ja siten myös eräänlainen Basic-tulkin manttelinperijä. Selaimen kaveriksi tarvitaan vain tekstieditori.

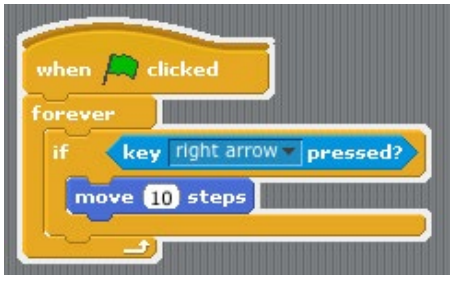
Moniin sovellusohjelmiin kuuluu jonkinlainen skripti- tai makrokieli. Mikäli siis esimerkiksi Excelin kanssa tulee vietettyä paljon aikaa, voi sen makrokielellä

leikkiminen olla luontevin ja hyödyllisin tapa saada ensikosketus ohjelmointiin.

Varo aivojen murjoutumista

Kun Basic yleistyi yliopistoissa, se herätti myös vastustusta. Havaittiin, että kieli opetti monet opiskelijat huonoille tavoille, joista he eivät päässeet irti. Ohjausrakenteiden alkeellisuus johti helposti ns. spagettikoodiin, jossa pahamaineisella goto-käskyllä hypittiin listauksen osasta toiseen. Tietojenkäsittelytieteilijä Edsger

Monen ensikosketus ohjelmointiin näytti tältä. Pahamaineinen GOTO opittiin siis jo heti toisena käskynä!



Hahmon liikutusta Scratch-kielellä.

Dijkstra totesikin jo 1970-luvun alussa, että Basic ”murjoo aivot korjauskelvottomiksi”.

Monet korkeakoulut ottivat Dijkstran näkemyksen tosissaan ja alkoivat kiinnittää erityistä huomiota ensimmäisen kielen ”puhtauteen”. Akateemiseen puhtauskäsitykseen sopivat erityisen hyvin Lisp-tyyliset funktionaaliset kielet, joita otettiinkin käyttöön alkeiskieliksi. Moni on törmännyt korkeakoulujen peruskursseilla esimerkiksi Scheme-kieleen.

Logo on monelle tuttu yksinkertaisena ”kilpikonnakielenä”, jossa tuotetaan kuvioita antamalla liikkumiskäskyjä kilpikonnahahmolle. Kieli ei kuitenkaan ole vain hauska lelu, sillä se kehitettiin johdatukseksi Lisp-ohjelmoinnin käsitemaailmaan. Kilpikonnan toi mukaan vuonna 1969 Seymour Papert, joka tunnetaan konstruktionismiksi kutsutun oppimisteorian kehittäjänä. Konstruktionismin perusajatuksena on, että oppija rakentaa sisäisiä malleja ympäröivästä maailmasta, esimerkiksi Logon kilpikonnan toiminnasta. Logo on ollut suosittu etenkin lasten ohjelmointikursseilla, mutta harvemmin kuulee edistyneempien ohjelmoijien antavan paljoakaan painoarvoa lapsuutensa Logo-kokeiluille.

Kokeneet ohjelmoijat ajattelevat usein, ettei aloituskielellä ole niin väliä. Huonojenkin lelujen kierouksista pystyy oppimaan irti, ja kykyä oppia uusia ajattelutapoja pidetään muutenkin hyvän ohjelmoijan ominaisuutena. On kuitenkin muistettava, että vain harvoilla on lahjoja tulla hyväksi ohjelmoijaksi, joten Dijkstran näkemystä ei kannata ohittaa aivan pelkällä olankohautuksella.

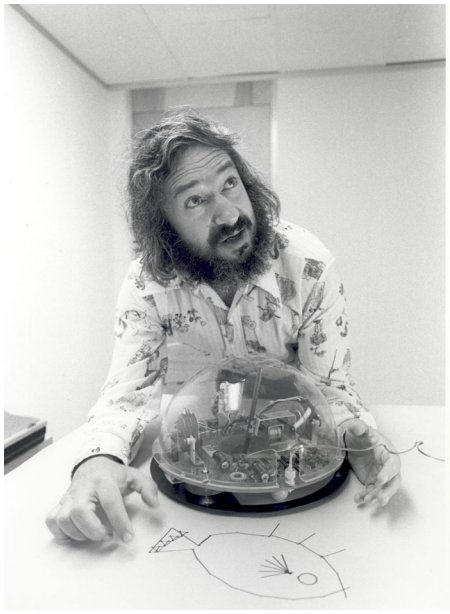
Useimmat nykyisin suositut opetuskäyttöön tarkoitetut kielet lienevät (basiceja lukuun ottamatta) melko turvallisia, vaikka monet niistä ovatkin akateemisen kuivakoita. Jos aivojen murjoutuminen pelottaa erityisen paljon, kannattanee pitäytyä vaikkapa vain Lisp- ja Smalltalk-pohjaisissa kielissä. Smalltalk perustuu varsin vahvasti Papertin oppimisteorioihin, ja esimerkiksi aiemmin mainitut lastenkielet Scratch ja Etoys ovat Smalltalk-pohjaisia.

Perinteinen tie: alhaalta ylös

Kun ohjelmointiin on saatu jonkinlainen tuntuma lelukielen avulla, on suurin vaikeus jo voitettu. Mutta kuinka päästä siitä eteenpäin?

1980-luvun kotitietokonemaailmassa teknisesti luontevin etenemistapa oli sukeltaa niin matalalle tasolle kuin mahdollista ja edetä siitä vähitellen korkeamman tason kieliin. Basicin jälkeen harrastaja tarttui assemblerin tai konekielimonitorin, jolla hän valjasti koneensa joka ikisen bitinliikkeen. Myöhemmin konekapasiteetin lisääntyessä siirryttiin C:n ja Pascalin kaltaisiin rakenteisempiin kieliin, joilla isommat projektit pysyivät paremmin hanskassa. Näistä jatkettiin myöhemmin olio-ohjelmointiin esimerkiksi C++:lla. Korkeamman tason rakenteita oppii ymmärtämään ja arvostamaan, kun on ensin joutunut toteuttamaan niiden tekemät asiat itse. Näin ne eivät jää pelkiksi muotoseikoiksi.

Tämä polku ei ole enää kovinkaan suosittu. Nykyaikaiset laitteet ovat huomattavasti monimutkaisempia kuin vanhat kotimikrot ja usein myös hautaavat bittinsä käyttäjän ulottumattomiin. Vanhankoulun polusta kiinnostuneen kannattaakin suosia nypläysvaiheessa yksinkertaisia alustoja, esimerkiksi klassisia kotimikroja, Arduinon kaltaisia mikrokontrollerilankkuja tai vaikkapa PC:n



Seymour Papert ja Logo-robotti vuonna 1973. Kuva: Susan Pogany, MIT Museum

DOS-käyttöjärjestelmiä. Emulaattorit korvaavat aidot laitteet hyvin, mikäli mukaan ei haluta liittää esimerkiksi rakenteluprojekteja.

Alhaalta ylös -polku sopii tee-se-itse-henkisille ihmisille, jotka haluavat ymmärtää järjestelmien toimintaa perusteellisesti ruohonjuuritasolta alkaen. Se tuppaa antamaan ohjelmointiin vahvasti konekeskeisen tuntuman, jossa koodin taloudellisuus menee inhimillisempien seikkojen ohi. Polkua kulkenut saattaa ahdistua hallitsemattoman monimutkaisista järjestelmistä ja haluta sellaisiin paneutumisen sijaan tehdä kaiken tarvitsemansa itse. Isojen kokonaisuuksien hallintaa saattaa joutua opettelemaan erikseen.

Kannattaa myös muistaa, että tämä polku on sellaisenaan monille melkoisen raskas. Mikäli konekieli ei kiinnosta heti lelukielen jälkeen, siihen ehtii kyllä syventyä myöhemminkin.

Toiset etenevät ylhäältä alas

Siinä missä jotkut haluavat rakentaa kokonaisuuksia alkeispalikoista, toisille on luontevampaa ottaa valmis kokonaisuus ja hajottaa se osiin. Ohjelmointitaitoa kehitetään paneutumalla valmiiden ohjelmien lähdekoodeihin ja muokkaamalla niitä. Taitojen ja ymmärryksen karttuessa pystytään tekemään itse aina vain monimutkaisempia palikoita ja ennen pitkää kokonaisia sovelluksia. Tämä polku lienee helpointa aloittaa esimerkiksi muokkaamalla HTML-sivuja tai skriptikielellä toteutettuja pikkuohjelmia.

Ylhäältä alas -lähestymistapa sopii etenkin käytännönläheisille ihmisille, jotka ovat uteliaita ohjelmien toiminnan ja rakenteen suhteen mutta joita kiinnostavat enemmän suuret kokonaisuudet kuin pienet nippelit. Polun kulkija saa hyvän käsityksen mahdollisista ohjelmien arkkitehtuureista, mutta edistyneempi algoritmiikka saattaa jäädä hänelle pimentoon.

Vaikka ”alhaalta ylös” ja ”ylhäältä alas” kuvataankin tässä erillisinä polkuina, ne kannattaa ajatella pikemminkin toistensa täydentäjinä kuin vaihtoehtoina. Kovapäisimmällekin tee-se-itse-henkilölle on hyödyksi oppia lukemaan ja sietämään myös muiden koodia, ja innokkaimmankin modaajan kannattaa välillä tehdä jotain myös täysin puhtaalta pöydältä.

Äkkiä suoraan ammattilaiseksi?

Jos aloittelija on erityisen tarkka päämääristään, hän saattaa päätyä pitämään alkeiskieliä ajanhukkana ja sivuuttaa ne. Hän on ehkä kuullut, että esimerkiksi C++ on tosiosaajien käyttämä kieli ja haluaa opetella suoraan sen. Jotkut ovat onnistuneet tässäkin, mutta lähestymistavassa on omat ongelmansa. Kielet ja ympäristöt, jotka ovat tehokkaita ja käytännöllisiä ammattilaisille, ovat usein jopa katastrofaalisia ohjelmoinnin opettelijoille.

Niille ihmisille, jotka haluavat päästä suorinta tietä päämäärään, on usein tärkeää vain saada ongelma ratkaistua eikä välttämättä edes ymmärtää ratkaisua. Nykymaailmassa tämä asenne johtaa herkästi hakukoneilla löytyneiden koodinpätkien käyttöön omien aivojen sijaan. Hakkerien Jargon-sanakirja käyttää tämäntyyppisestä ohjelmointitavasta halveksuvaa nimitystä ”cargo cult programming”.

Valmiiden ratkaisujen kopiointi on usein nopeaa, mutta entäpä kun vastaan tulee ongelma, johon ei ole valmista ratkaisua? Jos ohjelmoija ei ole harjoittanut omaa ongelmanratkaisukykyään vaan on valinnut aina helpoimman reitin, hän saattaa juuttua sormi suussa ihmettelemään yksinkertaisiakin pulmia.

Hakkerin kielivalinnat

Oppimispolun valinta on usein tasapainottelua käytännöllisyyden, opettavaisuuden ja ajankäytön välillä. Eric S. Raymondin opas ”How to Become a Hacker”

esittää hyvän kompromissin, jossa jokainen kieli on paitsi oppimisen kannalta tarkoituksenmukainen, myös käytännön tehtävissä ja hakkerikulttuurissa hyödyllinen.

Lelukieleksi Raymond suosittelee HTML:ää, joka ei ole ohjelmointikieli mutta antaa tuntumaa yleisempään tietokonekielten vaatimaan ajattelutapaan. HTML on myös nykymaailmassa äärimmäisen keskeinen kieli, jota on hyvä osata joka tapauksessa. Ensimmäiseksi ohjelmointikieleksi suositellaan Pythonia, joka on suhteellisen puhdas ja helppokäyttöinen mutta sopii myös laajempiin projekteihin. Pythonin jälkeen voidaan siirtyä opiskelemaan keskeisiä hakkerikulttuurin kieliä, joita ovat tehokkaan koodin mahdollistava C/C++, käytännöllinen Perl ja tärkeän valaistumiskokemuksen tarjoava Lisp.

Raymondin opas on tarkoitettu vapaan lähdekoodin hakkerikulttuuriin sisään haluaville, ja kielivalinnat ovat sen mukaisia. Hieman räätälöimällä ohjetta voinee kuitenkin soveltaa moniin muihinkin kulttuuriympäristöihin.

Yksin vai ryhmässä?

Ohjelmointia on perinteisesti opiskeltu joko koulukursseilla tai itsenäisesti, mutta nämä eivät suinkaan ole ainoat vaihtoehdot sosiaalisiksi puitteiksi. Joku oppii parhaiten yksityisopetuksessa, ja joku toinen saattaa kaivata ympärilleen pienryhmää. Monet ovat tykästyneet vuorovaikutteisiin opetusohjelmiin, jollaisena esimerkiksi Codecademy-sivustoa voidaan pitää. Puitteiden valinnassa kannattaa muistella omia kokemuksiaan erilaisten oppimistapojen toimivuudesta.

Vaikka varsinainen oppiminen tapahtuisikin itsenäisesti, voi sopiva yhteisö tai skene tarjota hyviä kannustimia opiskeluun. Kun sosiaalisessa ympäristössä on ohjelmoijia, myös oma kiinnostus ohjelmointia kohtaan säilyy. Jotkin yhteisöt järjestävät ohjelmointiin liittyviä kilpailuja, joihin osallistuminen voi motivoida aloittelijoitakin. On myös yksittäisten peli- tai ohjelmistoprojektien ympärille muodostuneita yhteisöjä, joihin liittyneistä ei-koodaajista tulee ennen pitkää projektin innostamana myös koodaajia.

Valitse polkusi!

Ohjelmoimaan voi oppia monilla eri tavoilla. Jollekulle toimii heti ensimmäinen ehdotettu tapa, kun taas joku toinen voi joutua yrittämään ja erehtymään montakin kertaa ennen kuin sopiva polku löytyy. Hyvä itsetuntemus saattaa helpottaa polun löytämistä – tai sitten ei. Kaikista ei tietenkään tule ammattiohjelmoijia tai edes kotitarveohjelmoijia, mutta jonkinlainen ohjelmointitaito auttaa joka tapauksessa käsittämään digitaalista maailmaa. Vaikkei pää tuntuisikaan taipuvan ohjelmointiin, kannattaa kuitenkin sinnikkäästi yrittää!

Oppivatko kaikki ohjelmoimaan?

Me Skrollissa haluaisimme uskoa, että kaikilla on edellytykset oppia ohjelmoimaan, kunhan sopiva menetelmä löytyy. Yliopistoissa ollaan asian suhteen kuitenkin skeptisempiä. Kokemus on nimittäin osoittanut, että huomattava osa esimerkiksi tietojenkäsittelytieteen maistereista ei osaa ohjelmoida, vaikka onkin läpäissyt pakolliset kurssit. Myös monet itseopiskelijat ovat huomanneet, ettei ohjelmointi yksinkertaisesti asetu päähän, vaikka sitä kuinka yrittäisi sovitella.

Sitä, kenellä on edellytykset ohjelmoinnin oppimiseen, on ollut yllättävän vaikeaa päätellä mistään etukäteen. Edellytysten olemassaolo ei nimittäin tunnu korreloivan esimerkiksi älykkyysosamäärän tai matemaattisen tai kielellisen lahjakkuuden kanssa. Tutkijat Saeed Dehnadi ja Richard Bornat löysivät kuitenkin vuonna 2006 jokseenkin toimivan tavan erottaa vuohet lampaista.

Dehnad ja Bornat antoivat ohjelmoinnin alkeiskurssin opiskelijoille ennen kurssin alkua tehtäväpaperin, joka sisälsi ohjelmointikieleen liittyviä tehtäviä: ”Mitkä ovat muuttujien a ja b arvot seuraavan koodin suorittamisen jälkeen?” Koska kurssilaiset eivät olleet ennen ohjelmoineet, he joutuivat arvaamaan vastaukset. Osoittautui, että parhaiten kurssilla menestyivät ne, joiden arvaukset noudattivat yhtenäistä logiikkaa – eli ne, jotka muodostivat ohjelmointikielen toiminnasta selkeän mielenmallin, oli se sitten kuinka virheellinen hyvänsä.

Tutkimuksen perusteella ohjelmoinnin oppimisen edellytyksenä olisi siis jonkinlainen kyky tai taipumus muodostaa mielessä proseduraalisia malleja erilaisista järjestelmistä. Voisiko tätä ominaisuutta jotenkin harjoituttaa ennen ohjelmoinnin oppimista? Mikäli useimmille ”vuohille” on mahdollista löytää tähän sopiva menetelmä, ei mikään ole enää esteenä todelliselle koko kansan tietokonelukutaidolle.

```
A 10 ZUK
B 20 ZUK
A B ZUK
A SPROOT
B SPROOT
```

```
A 10 ZUK
B A ZUK
A 20 ZUK
B SPROOT
A SPROOT
```

Mitähän nämä ohjelmat tulostaisivat? Mahdollisia konsistentteja vastauksia on monia. Kuinka monta löydät?

Prolog-kielen perusteet

Ohjelmointi voi olla muutakin kuin käskyjen latomista peräkkäin. Prolog-kielellä ohjelmoidaan loogisia sääntöjä ja todistellaan teoreemoja.

Teksti: Antti J. Ylikoski Kuvat: Mitol Berschewsky

Prolog-kielen nimi tulee ranskan kielen sanoista *programmation en logique*, suomeksi 'ohjelmointi logiikan keinoin'. Kieli syntyi vuosikymmeniä kestäneen logiikkaohjelmoinnin tutkimuksen ja käytännön kokemuksen tuloksena. Prolog on yksi vanhimmista logiikkaohjelmointikielistä ja niistä kaikkein tunnetuin.

Prologia on käytetty ensisijaisesti tekoälyn piirissä ja asiantuntijajärjestelmien ohjelmoinnissa. Niissä tietokoneohjelma pyrkii toimimaan kuin ihmisasiantuntija, joka tekee päätöksiä laajan tietämyksen ja kokemuksen perusteella. Prologia on käytetty myös luonnollisten kielten käsittelemiseen. Kieli on kuitenkin yleiskäyttöinen ja soveltuu moniin muihinkin tehtäviin.

Watson

Yhdysvalloissa tv-tietokilpailu Jeopardyssa oli yhtenä osanottajana Watson eli IBM:n laatima tietokoneen ja ohjelmiston yhdistelmä. Watson voitti tietokilpailussa ihmiskilpailijat, mitä pidettiin tärkeänä julkisuusansiona IBM:n ja tekoälyn kannalta. Watsonin tekoäly oli ohjelmoitu Prolog-kielellä.

Tämä artikkeli esittelee lyhyesti Prolog-kielen syntyhistorian sekä kielen keskeiset piirteet. Myös logiikkaohjelmoinnin alkeisiin tutustutaan.

Kielen synty

Prolog-kieli syntyi 1970-luvun alkupuolella. Sen keksijänä pidetään ranskalaista Alain Colmeraueria, jonka apuna työskenteli Philippe Roussel. He olivat kiinnostuneita luonnollisten kielten käsittelystä ja sovelsivat siihen automaattisia teoreemantodistustekniikoita.

Varhaisessa vaiheessa Prologin kehitykseen osallistuivat myös Jean Trudel ja Robert Kowalski, joiden avulla kehitettiin logiikkakielen päättelysääntöä, niin sanottua resoluutioalgoritmia. Itse algoritmi oli pääosin peräisin jo vuodelta 1965 ja perustuu Alan Robinsonin tutkimuksiin. Myös Robinson tutki logiikkaohjelmointia. Hän ymmärsi, että automaattisessa teoreemojen todistamisessa kannattaa erottaa toisistaan päättelysääntö (resoluutio) ja puutietorakenteiden samankaltaisuuden tutkimisalgoritmi (unifiointi).

Prolog-kielellä on lopulta hyvin monia kehittäjiä, ja siksi se onkin kehittynyt monin eri tavoin 70-luvun alun jälkeen. Kehittäjien yhteisenä tarkoituksensa on ollut logiikan soveltaminen ohjelman suoritukseen siten, että ohjelman suoritus voidaan nähdä teoreeman todistamisena.

Myöhemmin Prologista on pyritty tekemään yleiskäyttöinen logiikkaohjelmointikieli, jota voi käyttää suurten ja vaativien ohjelmistojen kirjoittamiseen. Prolog on siis levinnyt akateemisen maailman ulkopuolellekin, esimerkiksi teollisuuden ja kaupan aloille.

Uudet ajattelutavat

Tavanomaiset ohjelmointikielet toimivat käskyperiaatteella, imperatiivisesti. Ohjelman suoritus nähdään peräkkäisinä käskyinä, jotka muokkaavat ohjelman tai laitteiston tilaa. Niissä suoritetaan lauseita ja lausekkeita, sijoitetaan arvoja muuttujiin, määritellään ja kutsutaan funktioita ja niin edelleen.

Prolog-kielessä ja logiikkaohjelmoinnissa ohjelman suorittaminen nähdään teoreeman todistamisena. Ohjelmoija kuvaa logiikan keinoin käsillä olevan ongelman tai tehtävän. Hän määrittelee sääntöjä eli predikaatteja, jotka sisältävät tietoa asioiden välisistä suhteista. Predikaatit muodostavat tietokannan, joka toimii ikään kuin ohjelman tietoisuutena.

Kun tietokanta on valmis, ohjelmoija

vain pyytää ohjelmointijärjestelmää ratkaisemaan ongelmia. Ratkaisu syntyy automaattisesti, sillä kieli itsessään sisältää teoreemojen todistusalgoritmit eli tarvittavat resoluutio- ja unifiointialgoritmit. Ratkaisu perustuu ennalta määriteltyyn ongelmankuvaukseen ja sen loogisiin seurauksiin. Prolog-kieli soveltuukin hyvin tehtäviin, joissa asia voidaan ilmaista täsmällisin säännöin.

Teoreemojen todistamiseen voi liittää omia proseduureja. Tällä tavoin Prolog-ohjelmaan voi liittää lähes minkä hyvänsä itse ohjelmoidun toiminnon. Lisätoiminnot voi ohjelmoida toisillakin ohjelmointikielillä.

Prolog-ohjelman suorittama teoreemojen todistus voidaan kuvata ja suorittaa myöskin algoritmilla, jota kutsutaan peruuttavaksi hauksi tai etsinnäksi. Havainnollistan käsitettä shakkipelin avulla. Ajatellaan tilannetta, jossa shakkimestari miettii seuraavaa siirtoa. Tyypillisesti keskipeliasemassa on noin 30 mahdollista siirtoa, joista shakkimestari pitää kelvollisena ehkä kolmea tai neljää siirtoa. Hän suunnittelee siirtoketjua useita vaiheita eteenpäin: oma siirto, vastustajan siirto, oma siirto, vastustajan siirto, oma siirto jne. Koska vastustajan siirtoja ei voi täysin ennustaa, täytyy käsitellä monia vaihtoehtoisia tulevaisuuksia eli useita eri siirtovaihtoehtoja kuhunkin tilanteeseen.

Jossakin suunnitteluvaiheessa shakinpelaaja voi huomata: ”Hups, tuota miettimääni siirtoa ei voikaan tehdä, sillä se johtaisi tappioon.” Silloin pitää peruuttaa johonkin aiempaan suunnittelutilanteeseen ja miettiä muita vaihtoehtoja. Prolog-kielessä peruuttaminen merkitsee juuri tällaista palaamista aiempaan tilanteeseen ja sen jälkeistä muiden vaihtoehtojen kokeilua.

Prolog-järjestelmässä teoreemojen todistamista voi ohjailla mekanismilla, jota kutsutaan cut-primitiiviksi ja joka merkitään lähdekoodissa huutomerkillä. Kun yleisessä tapauksessa ryhdytään todistamaan teoreemaa, voidaan tehtävästä piirtää kuva, jota tietojenkäsittelyopissa kutsutaan puuksi. Kuva nimittäin muistuttaa jonkinlaista ylösalaisin käännettyä puuta. Teoreeman todistamisessa havaitaan melko usein, että osa puusta on tarpeeton tai jopa haitallinen suoritettavan tehtävän kannalta. Prolog-kielen cut-primitiivin avulla voidaan leikata turhia todistamispuun haaroja pois.

Tietotyypit

Ohjelmointikielten valtavirtaan verrattuna Prologin tietotyypeissä on jotakin tuttua ja jotakin erikoista. Lukutietotyypit ovat ihan perinteiset, eli tarjolla on kokonaisluku ja liukuluku. Liukulukujen tarkkuus on riippuvainen kielen toteutuksesta.

Varsinaista merkkijonotyyppiä ei standardi-Prologissa ole, mutta lainausmerkkien avulla voidaan kuitenkin luoda merkkijonoa muistuttavia objekteja:

```
"Tämä ei ole merkkijono vaan lista."
```

Tavallisesti listat ilmaistaan Prolog-kielessä hakasulkeilla, esimerkiksi `[1, 2, 3, 4, 5, 6]`, mutta lainausmerkkien avulla on kätevämpää kirjoittaa listoja, jotka sisältävät pelkkiä merkkejä. Tällaisia objekteja käyttämällä saadaan merkkijonoja vastaava toiminnallisuus.

Kielen standardissa ei ole taulukkotyyppiä, mikä johtuu eräistä muinaisista unifiointiin liittyvistä näkökohdista. Moderneissa Prolog-järjestelmissä kuitenkin on melkein aina myös taulukkotyyppi, mutta sen toteutus hieman vaihtelee.

Prolog on symbolinen kieli kuten Lisp – toisin kuin esimerkiksi Basic, Python, Perl tai Pascal. Kielen tietotyyppinä on symboli, mikä lisää kielen ilmaisuvoimaa ja tiedonkäsittelykykyä merkittävästi. Prolog-kielessä symbolia kutsutaan atomiksi. Niitä käytetään olioiden niminä, ja ohjelmakoodissa ne alkavat pienellä kirjaimella.

Logiikkaohjelmoinnin keskeisimmät objektit, predikaatit, koostuvat pääasiassa symboleista. Symbolinen tietojenkäsittely muistuttaa jossain määrin ihmisaivojen toimintaa, sillä aivotkin ovat jonkinlainen symboliprosessori. Symbolisuus soveltuu hyvin logiikka- ja tekoälyohjelmointiin.

Predikaattisymbolit ovat funktoreita. Funktorit ovat olioita, joilla on nimi (atomi) sekä joukko argumentteja. Argumentit ovat joko atomeja, funktoreita tai vakioita, kuten lukuja tai listoja. Seuraavassa on funktoriesimerkkejä:

```
isa(esko, antti)
fact(X, F)
super(X, funfun(Y))
sin(30, X)
yksikko(kalat, kala)
```

Prolog-ohjelmoinnin alkeet

Sukupuuesimerkki on tyypillinen tapa johdatella Prolog-kielen perusteisiin. Sukulaisuussuhteita voidaan ilmaista seuraavasti:

```
vanhempi(esko, antti).
vanhempi(esko, martti).
vanhempi(esko, anne).
```

Se ilmaisee, että objekti esko on objektien `antti`, `martti` ja `anne` vanhempi. Lisätään seuraavaksi samoille henkilöille myös äiti:

```
vanhempi(mailis, antti).
vanhempi(mailis, martti).
vanhempi(mailis, anne).
```

Tallennetaan edellä olevat koodirivit tiedostoon ”sukupuu.pl” ja käynnistetään sitten Prolog-järjestelmä. Näkyviin ilmestyy seuraavanlaisen komentokehote: ”`| ?-`”. Pystyviiva ilmaisee, että kyseessä on syöttöoperaatio, ja merkit `?-` liittyvät siihen, että ollaan todistamassa teoreemaa. Luetaan nyt muistiin Prolog-ohjelma, jonka aiemmin tallensimme:

```
| ?- ['c:\polku\sukupuu.pl'].
```

Hakasulkeet merkitsevät sitä, että ohjelma luetaan levyltä. Tiedostopolkua ei välttämättä tarvita, jos tiedosto sijaitsee nykyisessä työhakemistossa. Kun tiedosto on ladattu, voidaan suorittaa kyselyjä. Esimerkiksi seuraava kysely tulostaa kaikki olion `esko` lapset:

```
| ?- vanhempi(esko, Lapsi).
```

Kun argumenttina on muuttuja, esimerkiksi `Lapsi`, tulostuu kyselyssä kaikki kyseistä argumenttia vastaavat oliot. Prolog-kielessä muuttujat alkavat isolla kirjaimella ja vakiot pienellä kirjaimella, jos ne koostuvat yhdestä atomista. Seuraava kysely tulostaa kaikkien vanhempien kaikki lapset:

```
| ?- vanhempi(Vanhempi, Lapsi).
```

Lisätään vielä sukupuu.pl-tiedostoon seuraavat koodirivit ja luetaan jälleen ohjelman laajennettu versio Prolog-järjestelmään.

```
vanhempi(veli_matti, esko).
vanhempi(liisa, esko).
vanhempi(ville_matti, mailis).
vanhempi(silja, mailis).

% Tässä määritellään sääntö:
isovanhempi(X, Z) :-
    vanhempi(X, Y), vanhempi(Y, Z).
```

Edellisessä esimerkissä on jotakin aivan uutta: siinä määritellään sääntö.

Perinteisen tervehdysohjelman Prolog-toteutus

```
| ?- [user].  % Luetaan koodia päätteeltä.
terve :- write("Morjens, maailma!"), nl.
^D
| ?- terve.
Morjens, maailma!
```

Koodissa oleva `:-` on implikaationuoli, ja sääntö sanoo, että olio `X` on olion `Z` isovanhempi, jos `X` on `Y`:n vanhempi ja `Y` on `Z`:n vanhempi. Säännön määrittelyssä `vanhempi`-predikaattien välissä oleva pilkku on looginen ja-operaattori. Nyt voimme suorittaa esimerkiksi seuraavanlaisen kyselyn:

```
| ?- isovanhempi(Isovanhempi, Lapsenlapsi).
```

Prolog-järjestelmä tulostaa uskollisesti meille kaikki voimassa olevat isovanhempi–lapsenlapsi-suhteet. Ohjelmoidaan vielä lisää ja määritellään myös sukupuoli sekä isoisän sääntö:

```
mies(veli_matti).
mies(ville_matti).
mies(esko).
mies(antti).
mies(martti).
nainen(anne).
nainen(mailis).
nainen(liisa).
nainen(silja).

% Isoisän määrittelevä sääntö:
isoisa(X, Y) :-
    isovanhempi(X, Y), mies(X).
```

Uuden säännön mukaan `X` on `Y`:n isoisä, jos `X` on `Y`:n isovanhempi ja `X` on mies. Nyt esimerkiksi seuraava kysely tulostaa kaikki voimassa olevat isoisä–lapsenlapsi-suhteet:

```
| ?- isoisa(Isoisa, Lapsenlapsi).
```

Mennään seuraavaksi hieman pidemmälle. Kertomafunktion laskeminen on klassinen esimerkki ohjelmointioppaissa. Prolog-kielellä se näyttää seuraavalta:

```
kertoma(X, 1) :-
    X =< 1,
    !.

kertoma(X, F) :-
    Aux is X-1,
    kertoma(Aux, FAux),
    F is X * FAux,
    !.
```

Esimerkkiohjelman ylemmässä säännössä sanotaan, että luvun `X` kertoma on 1, jos `X` on pienempi tai yhtä suuri kuin 1. Oikeastaan tämä on matemaattisesti hieman väärin, mutta sen tarkoitus on varautua siihen virhetapaukseen, että käyttäjä antaa predikaatille argumentin, jonka arvo on negatiivinen. Nollan kertomahan on yksi.

Ohjelmassa jäljempänä olevaa sääntöä sovelletaan, mikäli aikaisempi ei toimi. Se sanoo, että luvun `X` kertoma on `F`, jos

1. apumuuttuja `Aux` saa arvon `X-1`
2. apumuuttujan `Aux` kertoma on `FAux`
3. muuttuja `F` eli tulos saa arvon `X * FAux`.

Molempien sääntöjen lopussa on !-merkki, joka on niin sanottu cut-primitiivi. Sitä käytetään teoreemojen todistuksen ohjailuun.

Nyt kertoma voidaan laskea tai tarkistaa esimerkiksi seuraavanlaisilla kyselyillä:

```
| ?- kertoma(5, Kertoma).
Kertoma = 120
| ?- kertoma(5, 120).
yes
```

Rakennetaan sitten vieläkin monimutkaisempi ohjelma ja kirjoitetaan tiedostoon seuraava koodi:

```
monikko(Sana, Monikko) :-
  atom_codes(Sana, Lista),
  atom_codes(t, KirjainT),
  append(Lista, KirjainT, MonikkoLista),
  atom_codes(Monikko, MonikkoLista), !.

yksikko(Monikko, Sana) :-
  atom_codes(Monikko, Lista),
  atom_codes(t, KirjainT),
  append(YksikkoLista, KirjainT, Lista),
  atom_codes(Sana, YksikkoLista), !.
```

Ohjelma toimii seuraavasti: Määritellään kaksi predikaattia, `monikko` ja `yksikko`. Predikaatti `atom_codes(Sana, Lista)` sanoo, että symbolin `Sana` kirjainten lista on listassa `Lista`. Esimerkiksi voisi olla `Sana = "talot"`, jolloin tulee olemaan `Lista = [t, a, l, o, t]`. Hakasulkeilla ilmaistaan Prolog-kielessä lista.

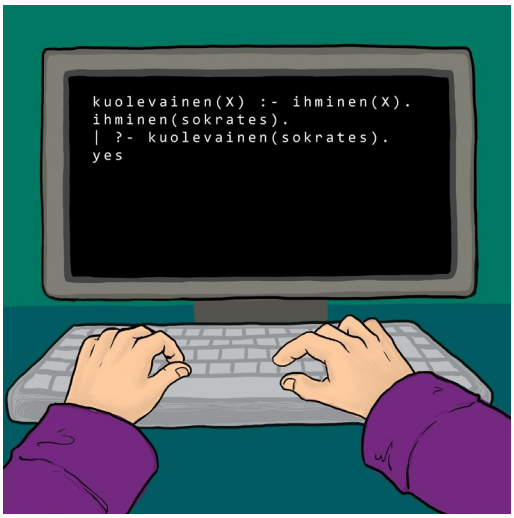
Nyt voidaan nähdä, että predikaatti `monikko` lisää t-kirjaimen sanan `Sana` loppuun ja predikaatti `yksikko` ottaa sanan `Monikko` lopussa olevan t-kirjaimen pois. Jos esimerkiksi olisi `Monikko = "autot"`, niin predikaatti `yksikko` laskee tulokseksi `Sana = "auto"`.

Prologia käytetään paljon luonnollisten kielten käsittelyyn. Tämä oli yksinkertainen esimerkki siitä, miten luonnollisen kielen merkityksellisiä yksiköitä eli morfeemeja voidaan käsitellä Prologilla.

Miten pääsen alkuun?

Mikäli Prolog ja logiikkaohjelmointi alkoi kiinnostaa, kannattaa aivan alkajaisiksi ladata netistä jokin ilmainen Prolog-järjestelmä. Hyviä ovat esimerkiksi kiinalainen BProlog, alankomaalainen SWI-Prolog sekä GNU Prolog. Kaupallisiakin toteutuksia on kyllä saatavilla.

Sen jälkeen tarvitaan hyvä oppikirja. Ivan Bratkon kirja *Prolog Programming for Artificial Intelligence* on mielestäni paras Prolog-tietolähde. Kannattaa hankkia uusin painos. Oppikirjoina klassikon asemassa ovat Richard O'Keefen *The Craft of Prolog* sekä Leon Sterlingin ja Ehud Shapiron *The Art of Prolog*. Hyvä opas on myös internetistä ilmaiseksi saatava Ulf Nilssonin ja Jan Maluszynskin *Logic, Programming and Prolog*.



Aristoteelisen logiikan Prolog-toteutus.

Jos haluaa tutustua Prolog-kielen hienouksiin syvemmin, Peter Norvigin kirja *Paradigms of Artificial Intelligence Programming* sisältää Lisp-kielellä toteutetun Prolog-tulkin sekä tarkempaa Prologin toteutuksen käsittelyä.

Prolog-kielestä voi keskustella esimerkiksi Usenetin ryhmässä comp.lang.prolog. Ryhmän aktiivikäyttäjät ylläpitävät muun muassa faq-listaa eli vastauksia usein kysyttyihin kysymyksiin. Siinä on tietoa lukuisista eri Prologin toteutuksista. Keskusteluryhmästä saa myös vertaistukea.

Kirjallisuutta

- Bratko, Ivan: Prolog Programming for Artificial Intelligence, 4th edition, Addison-Wesley/Pearson 2012, ISBN 978-0-321-41746-6.
- Nilsson, Ulf – Maluszynski, Jan: Logic, Programming and Prolog. Saatavissa: http://www.ida.liu.se/~ulfni/lpp/.
- Norvig, Peter: Paradigms of Artificial Intelligence Programming, Morgan Kaufmann 1992, ISBN 978-1-55860-191-0.
- O'Keefe, Richard A.: The Craft of Prolog, The MIT Press, ISBN 978-0-262-51227-5.
- Sterling, Leon – Shapiro, Ehud: The Art of Prolog, 2nd edition, The MIT Press, ISBN 978-0-262-69163-5.
- Tucker, Allen B. (editor): Computer Science Handbook, 2nd edition, Chapman & Hall / The Chemical Rubber Company 2004, ISBN 1-58488-360-X.

AMIGAAA!

SUOMEN AMIGA-KÄYTTÄJÄT RY.

SAKU 2014 | TAMPERE 20.9.

MITÄ
- SAKU 2014 -TAPAHTUMA JÄRJESTETÄÄN TAMPEREELLA PÄÄKIRJASTO METSOSSA LAUANTAINA 20.9.2014 KLO 10-16
- YHDISTYS PALAA TAMPEREELLE 10 VUODEN TAUON JÄLKEEN!

UUTTA
- UUSIMMAT KÄYTTÖJÄRJESTELMÄT JA YHTEENSOPIVAT KONEET: AMIGAOS 4.1, MORPHOS 3 JA AROS
- FPGA-KONEITA JA UUSIA LISÄLAITTEITA VANHOILLE KONEILLE
- PELIUUTUUKSIA KOKEILTAVANA

HISTORIAA
- USEITA ALKUPERÄISIÄ AMIGA-KONEITA
- MUITA 80-LUVUN AIKALAISIA VANHOISTA TUTUISTA HARVINAISUUKSIIN, COMMODORESTA SPECTRAVIDEOON

OHJELMAA
- ESITTELYJÄ, PELIKILPAILUJA, DEMOJA JA VAPAATA KÄYTTÖÄ

OSTOKSIA
- SUOMALAINEN AMIGA-JÄLLEENMYYJÄ GENTLE EYE KY
- TIETOKONEKULTTUURIN ERIKOISLEHTI SKROLLI
- YHDISTYKSEN TUOTTEITA

TAPAHTUMA ON ILMAINEN JA AVOIN KAIKILLE, TERVETULOA!

http://saku.amigafin.org | http://www.facebook.com/suomenamigakayttajat

HUOMIO!

Lukemalla Skrollia ilmaiseksi

RYÖSTÄT

tekijöiden omaisuutta!

Tilaa heti oma Skrolli:

http://www.skrolli.fi/tilaa

Bittikosmologia

Tekniikka on aina kiehtonut ajattelijoita. Vuosisatoja sitten todellisuus haluttiin nähdä kellokoneistona ja vähän myöhemmin höyrykoneena. Nykyisin ykkösehdokas kaiken malliksi on kuitenkin tietokone.

Teksti: Ville-Matias Heikkilä Kuvat: Ville-Matias Heikkilä

Tässä jutussa kerrotaan tietotekniikan inspiroimista todellisuuskäsityksistä. Tietotekniikan pohjalla on matemaattisia rakenteita, jotka keksittiin kauan ennen tietokoneita ja joihin on liitetty mystiikkaa jo alusta saakka. Kahdesta erilaisesta symbolista muodostuvat jonot ovat yksi näistä. Pääsemme siis aloittamaan tarinamme tuhansien vuosien takaa.

Bitit ennustajina

Ehkä tunnetuin ja vanhin bittijonoihin perustuva ennustusmenetelmä tulee Kiinasta. Menetelmässä arvotaan kuvio, joka koostuu kuudesta vaakasuorasta tikusta. Kukin tikku voi olla joko ehjä tai katkonainen. Lisäksi tikku voi olla siirtymävaiheessa päinvastaiseen tilaan. Muinainen ennustusopas nimeltä *I Ching* (Muutosten kirja) kertoo, kuinka kuuluu tulkita kukin 64 mahdollisesta kuviosta mahdollisine bittimuutoksineen.

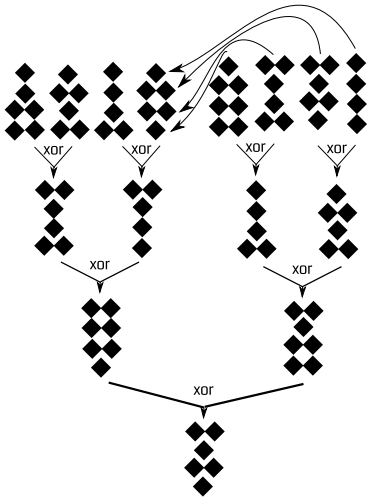
Bittiennustaminen vaikutti syvästi siihen, kuinka todellisuutta ruvettiin käsittämään kiinalaisessa filosofiassa. Maailmankaikkeuden ajateltiin olevan täynnä asioita, jotka muuttuvat aktiivisen *jang*-tilan ja passiivisen *jin*-tilan välillä. Monimutkaisempien muutosketjujen taakse miellettiin pitempiä bittijonoja. Esimerkiksi kuhunkin kolmen bitin kuvioon eli trigrammiin yhdistyy jokin kiinalaisten viidestä elementistä (tuli, vesi, maa, puu tai metalli).

Myöhemmin ilmestyi ennustusopas nimeltään *Tài Xuán Jīng* (Suurten ihmeiden kirja), jota voi pitää eräänlaisena trinäärisenä kilpailijana binääriselle I Chingille. Opas lisäsi jin- ja jang-tikkujen joukkoon kahdesti katkaistun tikun, joka kuvaa ihmisyyttä (*ren*) ja joka antoi merkityksen kullekin 81:lle neljän tritin yhdistelmälle.

Myös muualla maailmassa on harjoitettu binääriennustamista. Länsiafrikkalaiset järjestelmät käyttävät kuusibittisten yhdistelmien sijaan kahdeksanbittisiä, jotka esitetään kahtena vierekkäisenä nelibittisenä kuviona. Kuviot koostuvat yksittäisistä ja parittaisista kivistä. Yhdistelmä arvotaan heittämällä kahdeksasta simpukankuoresta koostuvaa opele-ketjua, jossa kukin kuori asettuu joko kupera tai kovera puoli ylöspäin. Järjestelmästä käytetään sentapaisia nimiä kuin *Ifa*, *Afa* tai *Fa*.

Bittien tulkintatavat eroavat toisistaan huomattavasti. Nigeriassa elävät igbot liittävät kuhunkin yhdistelmään vain yksinkertaisen sanan tai käsitteen, esimerkiksi *nähdä* tai *myrkky*. Jorubat puolestaan liittävät kuhunkin yhdistelmään pitkän tarinan, ja näin bittijonot toimivat indekseinä, jotka auttavat pitämään suullista perimätietoa kasassa. Gbe-kansat puolestaan antavat kahdeksanbittisen tunnuksen paitsi tarinoilleen, myös kasveille, eläimille, tapahtumille, ruokatabuille ja yleensäkin kaikille maailman ilmiöille.

Euroopassa ja Lähi-idässä tunnettu binääriennustusmenetelmä on nimeltään *geomantia* tai *ilm al-raml*. Geomantiassa arvotaan aluksi neljä neljän bitin sarjaa, joita pyörittämällä ja biteittäin yhteen laskemalla (xor) saadaan aikaan kartta, joka muodostuu 15:stä neljän bitin sarjasta. Kartan kohdat vastaavat elämän eri osa-alueita samaan tapaan kuin astrologiset huoneet. Geomantia jäi kuitenkin Euroopassa melkoisen harvinaiseksi menetelmäksi. Täällä oli jo Euklideen ajoista alkaen ihannoitu matemaattista jatkuvuutta, ja niinpä ennustajatkin seurasivat



Geomantiaa: oikean yläkulman neljä nelibittistä kuviota arvotaan, ja loput muodostetaan niistä pyörittämällä ja xor-operaatiolla. Kunkin kuvion merkitys riippuu sen koostumuksesta ja sijainnista.

Leibnizin hahmotelma binäärijärjestelmää juhlistavaksi medaljongiksi.

mieluiten planeettojen tasaisia liikkeitä taivaalla ja rakensivat salaoppinsa niiden pohjalle.

Binääriaritmetiikan uskonnollinen esiinmarssi

Nykyisen ykköseen ja nollaan perustuvan binäärilukujärjestelmän keksijänä pidetään saksalaista yleisneroa Gottfried Leibnizia (1646–1716), joka kaiken muun ohella kehitti myös differentiaalilaskennan ja rakensi yhden ensimmäisistä mekaanisista laskukoneista. Myös hän liitti binäärijärjestelmään uskonnollista mystiikkaa.

Leibniz oli ilmeisestikin saanut idean binärilukujärjestelmästä jo aiemmin, mutta hän innostui kirjoittamaan siitä vasta käytyään kirjeenvaihtoa jesuiittojen kanssa. Kiinassa lähetystyötä tehneet jesuiitat liittivät erääseen kirjeeseensä kuvan kahdeksasta I Chingin trigrammista, jotka Leibniz yhdisti oitis binärilukujärjestelmään. Se, että täysin vieras kansa oli toisella puolella maailmaa onnistunut löytämään saman matemaattisen periaatteen, oli Leibnizille merkki binäärijärjestelmän universaaliudesta. Koska kyseessä näytti olevan yksinkertaisin mielekäs lukujärjestelmä, varmastikin myös Jumala käyttäisi sitä. Kenties binääriluvuissa olisi salattuja merkityksiä, tai ehkäpä ykköset ja nollat olisivat koko kaikkeuden perustana.

Juutalaisessa mystiikassa on ajatus nimeltä *tzimtzum* ‘vetäytyminen’. Sen mukaan maailmankaikkeus oli alussa täynnä jumaluutta, joka tietyistä kohdista valikoidusti vetäytymällä antoi maailmalle muodon. Leibniz sovitti idean binäärijärjestelmään. Jumala eli ykkönen jättäisi vetäytyessään jälkeensä tyhjää eli nollia, joista maailma syntyisi. Leibniz innostui ajatuksesta niin paljon, että suunnitteli siitä kertovan hopeamedaljongin. Toteuttamatta jääneeseen medaljonkiin oli tarkoitus tulla taulukko binääriluvuista ja niiden desimaalivastineista, esimerkit binäärisestä yhteen- ja kertolaskusta sekä ajatus ”unus ex nihilo omnia” eli ”yksi [luo] tyhjästä kaiken”.

Luvuista rakentuva maailmankaikkeus oli kuitenkin Leibnizin aikaan jo vanha idea. Kun kreikkalaiset filosofit noin 2500 vuotta sitten kiistelivät siitä, olisiko maailmankaikkeuden peruselementti kenties vesi vai tuli, Pythagoras Samoslainen astui ihan kunnolla laatikon ulkopuolelle ja totesi sen olevan *arithmos* eli luku. Pythagoraan opetuksista lähteneet rönsyt vaikuttivat luonnontieteiden syntyyn mutta samalla ohjasivat ajattelua uralle, jota ruvettiin kunnolla kyseenalaistamaan vasta 1900-luvulla.

Elämmekö soluautomaatissa?

Kun tietokoneet yleistyivät tiedepiireissä, niitä haluttiin käyttää esimerkiksi maailman ilmiöiden simulointiin. 1960-luvulla nousi matematiikan ala nimeltä *symbolinen dynamiikka*, jossa maailman jatkuvia prosesseja pyritään mallintamaan soluautomaateilla ja muilla symbolijärjestelmillä. Aikakaudella vallinnut kyseenalaistava henki antoi pontta myös ajatukselle, jonka mukaan maailma ei olisi pohjimmiltaan jatkuva vaan diskreetti: symbolinen dynamiikka ei siis tarjoaisi maailmasta vain pikselöitynyttä likiarvoa, vaan todellisuus voisi oikeastikin koostua ”palikoista”. Esimerkiksi fyysikoiden aiemmin löytämät energiakvantit voisivat olla heijastumaa tästä diskreettiydestä.

Saksalainen tietotekniikkapioneeri Konrad Zuse kirjoitti vuonna 1969 kirjan *Rechnender Raum* (Laskeva avaruus), jossa hän esitti hypoteesin, jonka mukaan maailmankaikkeus olisi pohjimmiltaan kuin soluautomaatti tai muu tietokonemalli. Kirja loi perustan uudelle näkökulmalle, jota ruvettiin kutsumaan digitaalifysiikaksi.

Ehkä tunnetuin soluautomaatti on John Conwayn kehittämä *Game of Life*. Suoraviivaisimmillaan oma maailmankaikkeutemme voisi olla Game of Lifen kolmiulotteinen versio, jossa erilaiset paikalliset liiturien ja muiden kuvioiden rykelmät vastaavat esimerkiksi kvarkkeja ja välittäjähiukkasia. Soluautomaatin ei kuitenkaan tarvitse vastata havaittavaa fysikaalista maailmaa näin säntillisesti. Pohjalla voisi aivan hyvin olla vaikka yksiulotteinen Turing-täydellinen soluautomaatti, joka Turing-täydellisyytensä vuoksi pystyy samaan kuin kaikki kolmitai vaikka viisiulotteiset automaatitkin.

Jokainen tietokoneohjelma on universumi

Millainen laskentamekanismi maailmankaikkeuden pohjalla sitten olisi?

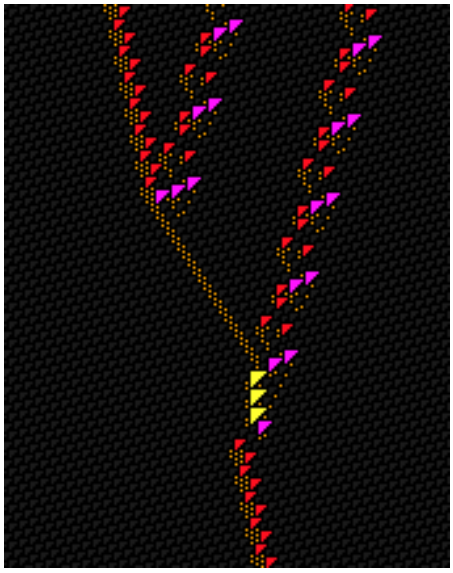
Leibniz oli sitä mieltä, että Jumala oli luonut parhaan mahdollisen maailman – sellaisen, joka synnyttää mahdollisimman paljon monimuotoisuutta mahdollisimman yksinkertaisista periaatteista. Jos tätä ajatusta viedään pidemmälle, voidaan päätyä maailmaan, johon kuuluvat kaikki mahdolliset maailmat. Voi hyvinkin olla yksinkertaisempaa kuvata periaate, joka tuottaa kaikenlaiset universumit luonnonlakeineen ja vakioineen, kuin kuvata vain yksi tietty vaihtoehto. Sama ajatus on esimerkiksi kosmologi Max Tegmarkilla, joka tunnetaan matemaattisen ja fysikaalisen olemassaolon samaistavista multiversumiteorioistaan.

Tekoälytutkija Jürgen Schmidhuber esitti vuonna 1997, että todellisuuden voisi kuvata pelkistettynä mutta muistiavaruudeltaan rajattomana tietokoneena, joka ajaa kaikkia mahdollisia ohjelmiaan samanaikaisesti. Vaikka suurin osa ohjelmista ei tekisikään mitään tolkullista, niiden joukkoon kuuluisivat myös kaikki mahdolliset maailmankaikkeuksien mallit, jotka siis toteutuisivat olemassa olevina maailmankaikkeuksina. Tämä ajattelutapa edustaa pankomputationalismia, jossa todellisuus ei ole pelkästään täysin mallinnettavissa ohjelmoitavalla tietokoneella vaan se pohjimmiltaan myös on sellainen.

Kaikki mahdolliset maailmankaikkeudet eivät kuitenkaan näytä olevan meille yhtä todennäköisiä asuinpaikkoja. Koska pitempiä ohjelmia on huomattavasti enemmän kuin lyhyempiä, älyllisiä olentoja ilmestyisi eniten sellaisiin maailmoihin, joista saa monimutkaisuutensa vuoksi hädin tuskin tolkkua. Luonnonlait olisivat täynnä omituisia poikkeuksia mutta kuitenkin riittävän säännönmukaisia kehittämään ja ylläpitämään älyllistä elämää. Meidän maailmamme ei nähtävästi ole tällainen, sillä sen luonnonlait tuntuvat tiivistyvän melkoisen ytimekkäiksi matemaattisiksi lausekkeiksi. Todellisuusselitykseen tarvitaan siis jotain, joka painottaa yksinkertaisempia malleja.

Ongelman ratkaisu voisi löytyä siitä, kuinka todellisuus jakaa laskenta-aikaansa eri maailmankaikkeuksille. Ohjelmia voitaisiin esimerkiksi ajaa sitä hitaammin, mitä pitempiä ne ovat. Jos perimmäistä todellisuutta ei haluta kuvitella ajalliseksi, voidaan laskenta-ajan sijaan ajatella, että lyhyempien ohjelmien tulosteet saavat osakseen ”enemmän olemassaoloa” kuin pitempien, ja sen vuoksi koemme todennäköisemmin juuri niitä.

Schmidhuber haluaa nähdä maailmankaikkeuden pohjalla perinteisen Tu-



Hiukkasentapaisten törmäilyä alkeissoluautomaatti 110:ssä. Satakymppi on Turing-täydellinen, joten se voisi teoriassa pyörittää meikäläistenkin todellisuutta.

ringin koneen, jolla on kuitenkin omat suorituskykyongelmansa. Kvanttimekaniikan kummallisuudet ovat melkoisen raskaita laskettavia Turingin koneen tapaan toimivilla koneilla, minkä fyysikot ovat simulaatioita tehdessään huomanneet. Kvanttifyysikko David Deutsch käyttää omassa mallissaan Turingin koneen kvanttiversiota, joka antaa paremmat toteutumismahdollisuudet erilaisille kvanttitason outouksille.

Simulaatiota simulaation päälle

Yksi tietojenkäsittelyn peruspilareista on Churchin–Turingin teesi, jonka mukaan kaikki tietokoneet ja tietojenkäsittelyn mallit pystyvät suorittamaan täysin samat tehtävät, mikäli muistitilaa on tarpeeksi. Tämä tarkoittaa esimerkiksi sitä, että erilaiset tietokoneet pystyvät emuloimaan toisiaan tarkasti. Digitaalifysiikassa se voisi tarkoittaa, että pienempiä maailmankaikkeuksia voidaan simuloida isompien sisällä. Simulaatio voi olla niin tarkka, ettei simulaatiossa elävä pysty mitenkään havaitsemaan, ettei hänen maailmansa ole itsenäinen universumi.

Transhumanistifilosofi Nick Bostrom esitti vuonna 2003 niin kutsutun simulaatiohypoteesin. Mikäli pienikin osuus ihmisen kaltaisista sivilisaatioista pääsee tasolle, jossa sillä on sekä valtavasti laskentakapasiteettia että kiinnostusta ”menneisyyssimulaatioita” kohtaan, se todennäköisesti ryhtyy toteuttamaan sellaisia. Vieläpä niin valtavissa määrin, että simuloituja elämiä on kaikkeudessa lopulta paljon enemmän kuin simuloimattomia.

Monet transhumanistit puhuvat teknologisesta singulariteetista, mutta kosmologi Frank J. Tippler menee vielä pidemmälle. Tipplerin mukaan universumi saavuttaa luhistuessaan kosmologisen singulariteetin, *omegapisteen*, jossa universumin laskentakapasiteetti paukahtaa äärettömäksi. Tällaisella laskentakapasiteetilla simuloituvat kaikki mahdolliset maailmankaikkeudet, ja kukin simulaatio voi kestää äärettömän pitkään.

Äärettömyys palauttaa meidät takaisin monimutkaisuusongelmaan. Jos olemme jo omegapisteessä, elämme todennäköisesti simulaatiossa mutta vielä todennäköisemmin simulaation sisäisessä simulaatiossa. Sisäkkäisiä tasoja on todennäköisesti aivan tolkuton määrä. Tarvitsemmeko kenties jonkin vielä vahvemman periaatteen, joka sijoittaa elämämme todennäköisemmin simulaatioiden ulko- kuin sisäpuolelle?

Voiko universuminsa valita?

Kaikkien mahdollisten tietokoneohjelmien joukossa on toki paljon muutakin kuin maailmasimulaattoreita. Esimerkiksi minkä tahansa elokuvan voi pakata ohjelmaksi, joka tulostaa sen. Mikäli kaikki erilaisten maailmasimulaattorien tuottamat maailmat ovat olemassa, pitäisi myös jokaisen elokuvan olla samaan tapaan olemassa itsenäisenä olionaan, joka ei tarvitse maailmankaikkeutta alleen.

Jos tietty elokuva halutaan pakata mahdollisimman lyhyeksi ohjelmaksi laitevaatimuksista piittaamatta, kannattaa ohjelmaan kuitenkin ympätä maailmasimulaattori. Mikäli elämme yksinkertaisessa maailmassa, simulaattoriohjelma on todennäköisesti lyhin mahdollinen ratkaisu. Ohjelma käytännössä simuloisi maailman, jossa elokuvantekijät syntyvät, elävät ja toteuttavat elokuvansa. Elokuvan valmistuttua simulaatiossa ohjelma lukisi sen solutaulukosta ja tulostaisi sen. Jos käytössämme olisi ääretön laskentakapasiteetti, ei meidän tarvitsisi edes suunnitella simulaattoria. Voisimme yksinkertaisesti vain käydä läpi kaikki mahdolliset ohjelmat lyhimmästä alkaen, kunnes vastaan tulee ohjelma, joka tulostaa elokuvan.

Simulaattorin ei tarvitsisi täysin vastata omaa universumiamme. Riittää, että itse elokuva tulostuu oikein. Tarkalleen saman elokuvan voi siis toteuttaa lukemattomilla eri tavoilla erilaisissa maailmankaikkeuksissa. Tämä avaa mielenkiintoisia näkökulmia myös ihmiselämän ja pankomputationalistisen todellisuuden suhteeseen.

Ihmiset vertaavat usein elämäänsä filmiin, joka sisältää kaikki elämän aikana koetut aistimukset, ajatukset, tunteet ja muut kognitiot. Myös tällainen elämänfilmi voidaan olettaa itsenäiseksi olioksi, joka on riippumaton maailmankaikkeuksista mutta joka voidaan kuvata monien mahdollisten universumien avulla. Voisi jopa sanoa, että elämme elämämme samaan aikaan kaikissa mahdollisissa universumeissa, joissa elämänkokemuksemme pystyy toteutumaan. Eri ihmisillä voi olla hyvinkin erilainen universumijoukko, eli jokaisella on oma yksilöllinen todellisuutensa.

Tietyissä älykköpiireissä suosioon ovat nousseet äärimmäisen relativistiset elämänkatsomukset, joissa todellisuus on paitsi yksilöllinen, myös yksilön tietoisesti valittavissa. Peter J. Carrollin kaaosmagia perustelee käsitystä kvanttimekaanisella multiversumilla, jossa tietoisuus etenee rinnakkaismaailmasta toiseen. Yhtä hyvin sitä voisi perustella myös pankomputationalistisen maailmankuvan kautta.

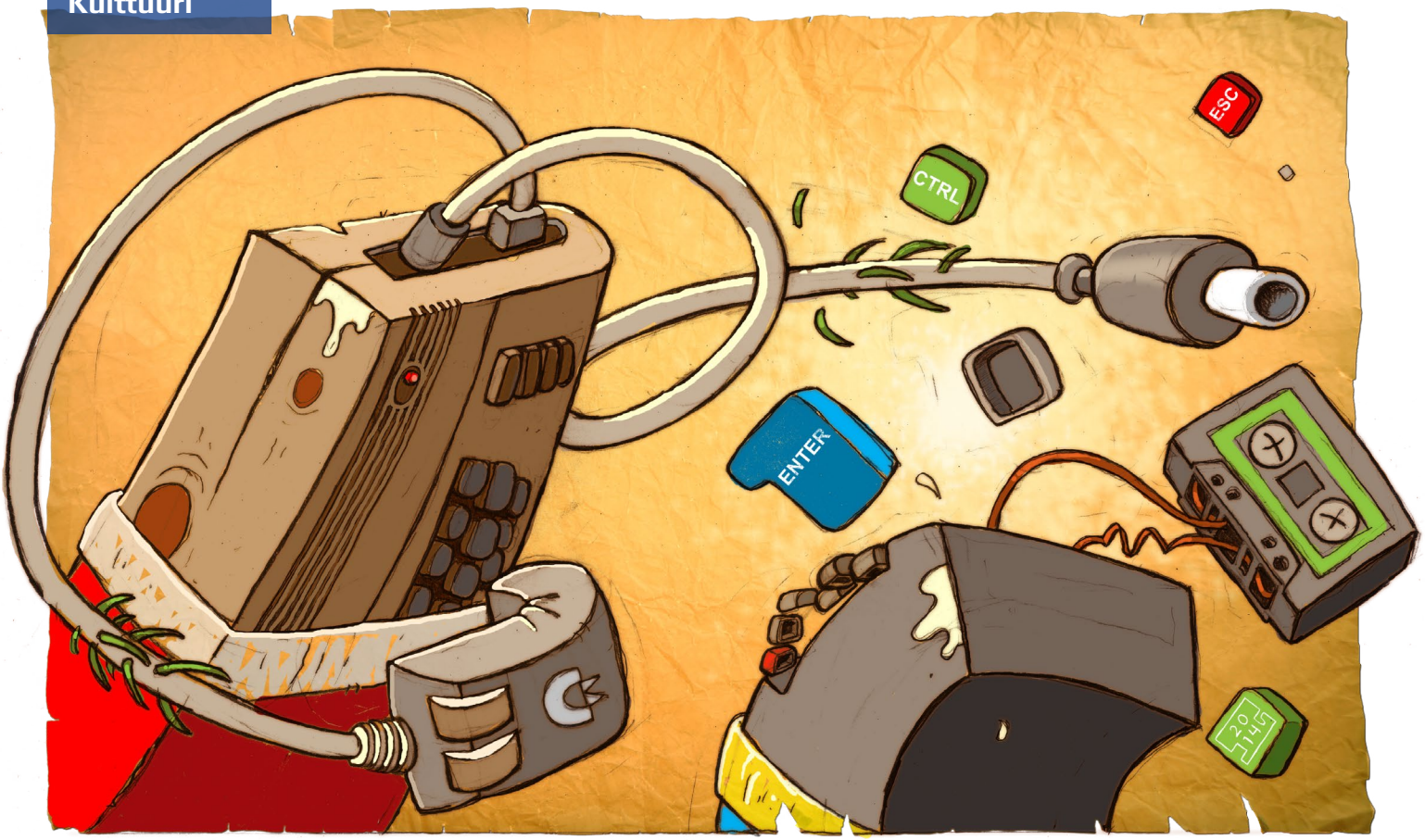
Onko tässä lopulta perää?

Tietokoneena kuvattu todellisuus vetoaa nykyihmisiin monin tavoin, ovathan digitaaliset virtuaalimaailmat arkipäivää. Simuloidun todellisuuden ajatusta on käsitelty valtavirtaviihteessäkin jo hyvän aikaa. Jos bittikosmologiset rakennelmat uppoavat jopa maallikoihin, saattavat ohjelmoijat ottaa ne vastaan suorastaan valaistuksenomaisesti. Oman alan teoria ei pädekään enää vain tietyillä sovellusalueilla vaan on avain koko todellisuuden ymmärtämiseen. Tiedepiireissä bittikosmologia on omiaan tuulettamaan luutuneita dogmeja ja avaamaan uusia näköaloja.

Kaikista valaistumiskokemuksista huolimatta kannattanee kuitenkin muistaa, että tietokonemetafora kertoo todennäköisesti enemmän meistä itsestämme ja nykykulttuurista kuin perimmäisestä todellisuudesta. Kun järjestäytynyt yhteiskunta oli uusi asia, haluttiin maailma nähdä eri asioihin erikoistuneiden jumalten ohjaamana. Kun höyrykone mullisti elämää, syntyi termodynamiikan kaltainen psykodynamiikka selittämään, kuinka ihmismieli käy kuumana ja päästelee höyryjä.

Parin sadan vuoden päästä saatamme jo naureskella sille, kuinka joskus uskoimme tietotekniikan olevan koko todellisuuden avain. Sitä, kuinka höyrypäät odottivat erilaisia singulariteetteja ja ties kuinka monennen sukupolven virtuaalitodellisuuksia kuin mitäkin uskovien ylöstempausta. Mutta älkäämme murehtiko myöhempien sukupolvien tuhahteluja nyt vaan nauttikaamme näköaloista, kun ne ovat vielä tuoreita!


Kesän paras viikonloppu on Assyillä
MESSUKESKUS, HELSINKI 31.7.-3.8.2014
WWW.ASSEMBLY.ORG
ASUS
IN SEARCH OF INCREDIBLE
JIMM'S
AXE



Tasavallan tietokoneen haastajat

80-luvun tietotekniikka kiteytyy monen mielessä Tasavallan tietokoneeseen, Commodore 64:ään. Se oli kaiken alku, se loi suomalaisen tietokonekulttuurin pohjan. Paitsi että ei ollut eikä luonut.

Teksti: Jukka O. Kauppinen

Kuvat: Tapio Lehtimäki, Wikimedia Commons

Commodore 64 oli tärkein ja tunnetuin laite, jolla Suomea digitalisoitiin, mutta ei ainoa eikä ensimmäinen. Kuusnelonen, Neba, nousi suomalaisen tietokonekulttuurin lipunkantajaksi vasta monivuotisen kamppailun jälkeen. Suomalaisten tietokoneenkäyttäjien sieluista taisteltiin ankarasti ja kentällä oli monta tasaväkistä haastajaa. Lopulliset rintamalinjat rakentuivat vasta vuosien kuluessa.

1980-luvun alussa kotitietokonemarkkinoille rynnisti kymmeniä uusia valmistajia. Mikrot nähtiin lupaavana kasvualueena, etenkin pelikonsolibisneksen jo näivettyessä ylitarjonnan alle. Osa uutuuksista saapui myös Suomeen, ja meilläkin myytiin yhtä aikaa muun muassa japanilaisia, amerikkalaisia, brittiläisiä, ruotsalaisia ja jopa suomalaisia kotitietokoneita. Alkuaskeliaan ottavat suomalaiset tietokonelehdet ja Tekniikan Maailma esittelivät liki joka numerossaan jonkin uuden tietokoneen, eivätkä koneet tietenkään olleet keskenään yhteensopivia.

Mikrokenttä olikin vahvan monikulttuurinen, eikä yksikään kone ollut selvästi toista parempi, vaikka pikkuasioista kiisteltiin ankarasti. Koneet olivat tekniikalta lähes identtisiä, muistin määrää ja prosessoria myöten. Loppumetreillä ratkaisevia valtteja olivat lopulta ohjelma- ja pelitarjonta. Joidenkin mikrojen ohjelmatarjonta oli laitteen valmistajan ja Mikrobitin ohjelmalistausten varassa. Toisille pelejä sai niin kaupoista kuin kaverilta kopioimalla.

Skrolli sukeltaa historian uumeniin ja muistelee 80-luvun konesotien unohdettuja sotureita, niitä osapuolia, joille ei ole rakennettu muistopatsaita. Häviäjiä, joita ei tänään muisteta konesotien sankarihaudoilla. Niitä mikroja, jotka antoivat panoksensa suomalaisen tietotekniikan kehittymiselle ja tekivät meistä sitä, mitä me tänään olemme.

Artikkelissa ei edes yritetä käsitellä aihetta tasapuolisesti ja kaikenkattavasti. Skrolli kuitenkin jatkaa aiheen käsittelyä myöhemmin verkkosivuillaan.

Aikajana

1977 Apple II
1977 TRS-80
1978 Atari 400/800
1979 Atari 800
1980 Commodore VIC-20
1980 Sinclair ZX80
1981 Sinclair ZX81
1982 Commodore 64
1982 Dragon 32
1982 Sharp MZ
1982 Sinclair ZX Spectrum
1982 Telmac TMC-600
1983 Atari 800XL
1983 Memotech MTX512
1983 Oric 1
1983 Salora Fellow
1983 Salora Manager
1983 Sega SC-3000
1983 Spectravideo SVI-328
1983 MSX-standardi
1983 Sony HitBit MSX
1984 Amstrad CPC464
1984 Oric Atmos
1985 Spectravideo SVI-728 MSX

Amstrad CPC464 "Sitkeä britti"	
Julkaisuvuosi	1984
Prosessori	Zilog Z80A, 4 MHz
Muisti	64 kt, laajennettavissa 576 kt
Näyttö	160×200 16 väriä; 320×200 4 väriä, 620×200 2 väriä
Äänipiiri	kolmekanavainen AY-3-8912-piiri kohinageneraattorilla

Elektroniikkayhtiö Amstrad rynnisti kotitietokonemarkkinoille CPC464-kotitietokoneella vuonna 1984 ja loikin huippumikron, joka kesti markkinoiden kulutusta hämmästyttävän pitkään. CPC suunniteltiin Amstradin ydinfilosofian mukaisesti yhtenäiseksi kokonaisuudeksi. Näppäimistö, kasettiasema ja varsinainen tietokone olivat samassa kotelossa. Jokainen kone myytiin monitorin kera, ja paketista tuli ulos ainoastaan virtajohto. Tämä olikin yksi koneen valteista: se oli helppo ottaa käyttöön: virtapiuha seinään ja menoksi!

Kone oli vakuuttava, näyttävä ja luotettava työkalu. Levyaseman formaatiksi kuitenkin valittiin oudot kolmituumaiset levykkeet. Hyvään varusteluunsa suhteutettuna CPC oli suhteellisen edullinen, mutta koneen hinta oli kuitenkin riisutumpia kilpailijoitaan korkeampi. Niinpä moni valitsi ennemmin halvemman vaihtoehdon ja kytki sen halpaan matkatelevisioon.

CPC-mallien valmistus jatkui aina vuoteen 1990 saakka. Lisäksi Amstrad ehätti ostamaan Sinclair Research -yhtiön Spectrumeineen vuonna 1986, minkä myötä firma voidaan laskea 80-luvun mikrosotien voittajiin.

Tekniikka

CPC464 oli yksi aikansa tehokkaammista koneista, jossa oli hyvä äänipiiri, nopea prosessori ja kohtuullinen grafiikka. Videopiirissä ei kuitenkaan ollut hardwarespritejä, mikä oli iso ongelma pelikehittäjille.

Kaikki amstradistit eivät olleet tasaarvoisia. CPC:n saattoi ostaa joko väri- ja vihermustanäytön kera, ja jälkimmäinen päätyi edullisempana vaihtoehtona moneen kotiin. Niinpä kaikki eivät koskaan nähneet pelejään väreissä.

Koneen nopeasta ja helppokäyttöisestä Basicistä pidettiin yleisesti.

Pelit

Amstrad panosti pelipuoleen alusta saakka ja oli tiiviissä yhteydessä pelitaloihin. Peleistä ei ollut pulaa jahka kone osoittautui hitiksi. Ne kuitenkin kärsivät samasta pulmasta kuin Amiga ja Atari ST myöhemmin: pelit olivat suurelta osin käännöksiä Spectrumilta. Niinpä CPC:n kyvyt jäivät usein hyödyntämättä, eivätkä pelikehittäjät tehneet sille juurikaan alkuperäisteoksia.

Joihinkin käännöksiin kuitenkin panostettiin enemmän, joten Amstradin Speccyä parempi grafiikka ja äänet pääsivät oikeuksiinsa. Esimerkiksi Head Over Heels, Get Dexter, Gryzor ja Spindizzy ovat Amstradilla parhaimmillaan. Muita oivallisia Amstrad-julkaisuja olivat North & South, The Guild of Thieves, Fantasy World Dizzy ja jopa Prince of Thieves.

Vaikka viimeinen CPC valmistettiin vuonna 1990, tehtiin laitteelle kaupallisia pelejä vielä 1993.

Suomessa

CPC:stä tuli yksi Euroopan suosituimmista kotitietokoneista. Britannian lisäksi se oli huippusuosittu myös Ranskassa ja Espanjassa – ja Suomessa. Amstrad-harrastajat ehtivät järjestäytyä kerhoksikin. Koneen hinta oli kuitenkin iso kynnys: vihernäytöllä kone maksoi 3980 markkaa, värinäytöllä 5980, eli tuplasti C64:ään verrattuna.

"Vaarini osti aikanaan Amstrad CPC6128:n levykeasemalla ja vihermustanäytöllä. Ensimmäiset ohjelmointikokemukset hankin Basicilla, tein jotain äärimmäisen hauskoja kyselypelejä ja ylhäältä alas skrollaavan autopelin tekstigrafiikalla. Tekstiseikkailussa yritin huijata printtaamalla ohjelmakoodin, mutta kiero koodari oli tehnyt siitä sellaista gotospagettia, ettei sitä kautta saanut apua peliin." ***-Shrike***

"Kiva kampe, oma monitori ja nopea kasettiasema. Mutta eipä noita kaveripiirissä kovinkaan montaa ollut, joten softien saatavuus oli mitä oli. Basiciin ja firmwareen tutustumisen jälkeen se oli mielenkiintoisin kapistus tehdä jotain muutakin kuin purkaa pelien suojauksia." ***-JSA***

"Paikallisesta infosta mukaan tarttui CPC6128. Jälkeenpäin huomasin, ettei kellään kaverilla ollut samanlaista konetta. Kaikki pelit piti ostaa ja niidenkin saatavuus oli heikohko. Amstrad kärsi suorista Spectrum-käännöksistä ja usein kaverin kuuslankulla hyvä peli oli omalla koneella tylsä tekele. Valopilkuista parhaimmin jäivät mieleen Head Over Heels ja Academy (Tau Ceti II). Vanhempien kauhuksi löytyi Barbarian, jonka mukana tullut juliste koristi pitkään huoneen seinää. Koulun välitunnit väiteltiin omien koneiden parhaimmuudesta ja fanaattisuus muistutti urheilujoukkueen kannatusta. Koneen heikkouksista huolimatta on aika kullannut muistot ja niitä tulee edelleen verestettyä WinApen kanssa."
–Jani "Cauldron" Valtonen

MSX ”Japani haastoi länsimikrot”	
Julkaisuvuosi	1983
Prosessori	Zilog Z80A, 3,58 MHz
Muisti	8-512 kt RAM, 16 kt näytönohjainmuisti, 32 kt ROM
Näyttö	512×212 pikseliä, 16 väriä; 256×212 pikseliä, 256 väriä
Äänipiiri	kolmekanavainen AY-3-8910-piiri kohinageneraattorilla

Suomi on siitä kumma maa, että japanilainen MSX-tietokonestandardi nousi meillä kotimikrojen ykkössarjaan. Kotimaassaan supersuositusta järjestelmästä nimittäin tykkäsivät meidän lisäksemme lähinnä hollantilaiset. Britanniassa sitä jopa parjattiin, sillä japanilainen mikro oli epäisänmaallinen valinta.

MSX ei ollut mikään yksittäinen tietokone vaan standardi. Jokaisesta MSX:stä löytyi sama käyttöjärjestelmä ja Microsoftin Basic, prosessori, äänipiiri sekä vähintään kahdeksan kilotavua muistia. Valmistajat saivat muokata omia mallejaan mielensä mukaan minimiä paremmaksi.

Valmistajia oli parhaimmillaan kymmeniä, joten valikoima oli laaja ja monipuolinen. Edullisimmat koneet riittivät peli- ja harrastuskäyttöön, mutta esimerkiksi Yamaha näki koneensa osana laajempaa musiikillista tuoteperhettään. MIDI-liitännöillä ja syntetisaattoriohjelmistoilla varustetut koneet olivatkin muusikoiden mieleen.

MSX sinnitteli useimpia kilpailijoita pidempään, sillä alkuperäisen MSX:n jatkajaksi esiteltiin vuonna 1986 MSX2, 1988 MSX2+ ja vielä 1990 MSX Turbo R.

Tekniikka

MSX:n prosessori oli nopeampi kuin esimerkiksi ZX Spectrumissa, ja merkittävästi C64:ää ripeämpi. Myös äänipiiri oli laadukas, vaikkakaan ei C64:n SIDin tasoinen. Peruskoneetkin olivat tasapainoisia ja laadukkaita, ja teknisesti parempia kuin Tasavallan tietokone.

MSX:t kuitenkin kärsivät muutamista ongelmista: etenkin videopiiri oli vaikea ohjelmoitava. Vaikka Spectrum-pelit kääntyivät helposti MSX:lle, ne eivät välttämättä yltäneet brittimikron tasolle.

Pelit

Laitteen perintö elää edelleen peleissä. Etenkin japanilaiset loivat koneelle runsaasti aikansa parhaimpia pelejä. MSX:llä saivat alkunsa muun muassa Castlevania-, Metal Gear- ja Bombermanpelisarjat. Muita aikansa huippuja olivat Contra, Dragon Quest, R-Type, Ys, SD Snatcher ja Gradius.

Kaikissa MSX-koneissa oli yksi, joskus kaksikin moduuliporttia, joten osa peleistä myytiin moduuliversioina.

Suomessa

MSX:llä ei ollut Suomessa yhtä virallista edustajaa, vaan kukin valmistaja edusti laitteitaan itse. Niinpä MSX-koneita saattoi löytää niin musiikki-, konttorikone-, kodintekniikka- ja joskus jopa tietokoneliikkeistä.

Spectravideo-koneiden maahantuoja Teknopiste oy kunnostautui myös MSX:n edustajana ja julkaisi MSX-pelejä, samoin tamperelainen Triosoft oy. Näistä tunnetuin lienee puolimyyttinen sotastrategia Talvisota. Moni julkaisi lisäksi pelejään indie-henkisesti.

MSX oli Suomessa ihmeen suosittu ja yleinen laite, joka sinnitteli C64:n kainalossa vielä pitkään muiden kilpailijoiden kadottua.

”Olin kodinkoneliikkeen huollossa työharjoittelussa. Pomo tuli sinne risojen joystickien keskelle erikoisen pyynnön kanssa. MSX:lle piti koodata jotain myyntiä edistämään, sillä yhtään konetta ei oltu myyty. Sovelsin sitten Sega SC-3000-koneen autopelin MSX Basicille – ja ainakin se esittelykone myytiin.” ***-Sunn***

”Valitsin SVI-728:n ylivertaisen Basicin takia. Etupäässä harrastinkin sillä Basic-ohjelmointia. MSX:lle oli loistavia pelejä, erityisesti Konamin moduulit. Erityisin hetki lienee ensimmäinen oma peli, jonka tein innoissani joululomalla heti koneen saatuani. ” ***-Wipu***

”MSX:ää hehkutettiin maailmanlaajuisena standardina. Myöhemminkin hehkuttelin sen MSX-DOSia ja CP/M-yhteensopivuutta, mutta yhtään moista ohjelmaa en nimittäin osaisi nimetä, tuskinpa siis käytinkään. Omia pelejä tuli koodailtua hyvällä Basicilla, ja tunsinkin suurta ylpeyttä, kun joku kavereista tuli meille pelaamaan niitä.” ***-oo***

”Isä vaihtoi paikallisessa kodinkoneliikkeessä maalaamansa öljyvärityön Sony HitBit MSX:ään. Yhtään peliä ei saatu jouluaattona latautumaan, mutta yöllä isä herätti mut, että ’NYT LATAUTU’. Peli oli Chiller. Nemesikset oli mun lemppareita. Meidän kylillä ei ollut kauheasti konesotia. Tunsin vähän ylpeyttä siitä, että olin erilainen, vaikka jotkut jutut nepalla olivatkin siistejä.” ***-Manu Pärssinen***

”Kone oli Sonyn HitBit kahdella moduuliportilla. Konamin peleihin sai kaikkea jännää, kun yhdisteli pari pelimoduulia. Triopostissa oli muistaakseni ohjeita ja koodeja. Basicilla tein kaikenlaista peliohjelmoinnista hyötyohjelmiin. Mikrobitistä naputeltiin todella paljon pelilistauksia. Omia räpellyksiä olivat kaverin kanssa kehitetty purjehduspeli, jonkin sortin strategiapelin runko ja Mikrobitin ’Näin rakennat tekstiseikkailun’ -artikkelin innostamana oma seikkailupeli. Moduuleista lempipelejäni olivat Nemesis-sarja, Salamander, Maze of Galious, King’s Valley ja F1-Spirit. Kotimaisista peleistä mieleen jäivät Triosoftilta tilatut Talvisota ja Roller.” ***-Nexus-6***

ZX Spectrum	
Julkaisuvuosi	1982
Prosessori	Zilog Z80A, 3,5 MHz
Muisti	16 tai 48 kt RAM, 16 tai 48 kt ROM
Näyttö	256×192 pikseliä, 8 väriä
Äänipiiri	yksikanavainen sisäänrakennettu kaiutin

ZX Spectrum oli Sinclairin kolmas ja merkittävin kotimikro. Vuonna 1980 julkaistu ZX80 ja vuotta myöhemmin seurannut ZX81 olivat ensimmäiset Isossa-Britanniassa myydyt alle sadan punnan hintaiset kotitietokoneet, ja ne opettivat ensimmäisen brittisukupolven tietokoneiden saloihin. Kolmannesta Sinclairista haluttiin jotain pysyvämpää.

Brittimiljardööri Clive Sinclairin rakkaimmasta lapsesta kasvoikin aikansa suosituin mikro, joka oli halpa ja sympaattinen. Suuri osa 80-luvun alkupuolen brittiläisistä klassikkopeleistä tehtiin ensin Spectrumille ja vasta sitten muille alustoille. Neuvostoliitossa ja muissa Itä-Euroopan maissa laitteesta tehtiin kymmeniä klooneja, joiden ympärillä elää yhä suomalaista C64-skeneä vastaava harrastajakulttuuri.

Spectrum olisi voinut olla hieman nopeampikin, mutta tarkkavainuinen bisnesmies kun oli, herra Sinclair mietti jokaisen kustannuksen tarkoin. Niinpä koneen kehitystä ei viety aivan niin pitkälle kuin olisi ollut mahdollista.

Mikro muistetaan ennen kaikkea edullisena pelikoneena, ja syystäkin. Hyötykäyttöön koneesta ei ollut. Retrosedät saattavat muistaa Spectrum+:n räikeän Pomo-mainoksen, jossa poika kehuu hankkineensa tietokonejengiin liittyessään kunnon koneen, jonka kanssa ei tarvitse räpeltää.

Tekniikka

ZX muistetaan oudonvärisistä peleistään ja piipittävistä äänistä. Väripaletti oli rajallinen, eikä sisäänrakennettu yksikanavainen kaiutin musiikillisesti juhlinut. Prosessori oli kuitenkin Commodore 64:ää nopeampi ja soveltui paremmin vektorigrafiikkaan. Koneen yksinkertaisuus tosin aiheutti peliohjelmoijille hienoisia haasteita.

Monille mieleenpainuvin muistikuva Specusta liittyy näppäimistöön. Koska koneen hinta oli kaikki kaikessa, varustettiin ZX kumimatosta tehdyllä näppäimistöllä, joka soveltui kehnosti kirjoittamiseen. Niinpä kaikki peruskomennot ja jopa Basic-ohjelmointiin tarvittavat komennot piti antaa näppäinoikoteillä. Hyvä niin, sillä mikron uumenissa piili helppokäyttöinen ja tehokas Basic-ohjelmointikieli.

Ajalleen tyypillisesti Spectrum käytti massamuistina kasettiasemaa, mutta sitä ei välttämättä myyty koneen mukana. Mikroon pystyikin liittämään minkä tahansa kasettisoittimen. Joystick-liitintä ei ollut, joten puolipakollinen tikkulaajennus nosti koneen hintaa.

Pelit

Spectrumille julkaistiin kaikkiaan yli 20 000 peliä. Laitteen tähtikehittäjiin lukeutuivat muun muassa Ultimate, josta kehittyi myöhemmin rakastettu Rare. XCOM-pelien luoja Julian Gollop teki ensimmäiset pelinsä Spectrumille, ja koneella nähtiin myös Lords of Midnight -sarjan luonut Mike Singleton ja Jeff Minter. Spectrumille tehtiin ensimmäiset isometriset ja kolmiulotteiset pelit, joskin moni muistaa parhaimmin koneen laadukkaat tasoloikat ja seikkailut. Tarjolla oli myös lentosimulaattoreita ja avaruusmatkailuja, eikä koneella ollut koskaan puutetta laadukkaasta pelattavasta.

Vaikka rajallinen väripaletti teki peleistä visuaalisesti erikoisia, niin etenkin viiva- ja vektorigrafiikkapeleissä Spectrum oli ylivertainen.

Suomessa

Spectrum oli pitkään yksi suosituimmista kotimikroista, jolle riitti runsaasti käyttäjiä ja faneja.

”Ostin kielikurssilta Englannista kesätyörahoillani ZX Spectrum 16 -kotimikron. En uskaltanut ostaa 48 kilon versiota, koska pelkäsin jääväni kiinni tullissa. Jos muistan oikein, niin alle 1 000 markan koneen sai tuoda tullitta käsimatkatavaroissa.” ***-saro30***

”Kaksinpelit olivat näppäimistöllä vähän ahtaita, mutta hauskoja. Kuminäppis oli muuten pirun näppärä Basicin kirjotuksessa, kaikki komennot kun sai shifteillä yhdeltä näppäimeltä. Kutsuimme hyvin opittua nopeaa kirjottelua krampaamiseksi. Pelaaminen ei ollut meille ainut juttu, vaan teimme kovasti ohjelmia ja kaikenlaisia hauskoja kokeita. Kaveripiirissämme useimmat harrastivat myös ohjelmointia. Itse tein muun muassa taustamusiikkia soittavan konekieliohjelman.” ***-pevri***

”48+ oli oma kone, kunnon näppiksellä. Tällä tuli kirjoitettua koodia, matopelejä, rivin demo-ohjelmia ja harjoitettua kräkkeröintiä. 48+ oli paras ystävä 80-luvun ajan, ja Your Sinclair -lehdet luettiin hartaasti kannesta kanteen sanakirja toisessa kädessä.” ***-ArhPos***

”Spectrumin puolella avautui pelien ihmemaa. Tekstiseikkailuja rakastin yli kaiken. Englannin kielen taitoni kiittää, sillä ilman tv:n Näkemiin vaan murua ja Spectrumin tekstipelejä en puhuisi mitään vieraita kieliä. Oli muuten pelien vaikeusaste ihan toisenlaista luokkaa kuin nykyään, jolloin joku Dark Souls on muka vaikea.” ***-Rasimus***

”Kokeilin ohjelmointia Basicillä ja konekielellä. Spectrumin mukana tuli käsikirja, missä oli konekielisten käskyjen lyhenteet, kuten MOV A,B tai sinnepäin, ilman sen kummempia selityksiä. Oli hauskaa arvailla. Spectrumilla on hauskaa ja mielekästä touhuta. Kovina peli-iltoina nostimme koneen jäähtymään ikkunan ulkopuolelle silloin tällöin. Jos ei satanut.” ***-Ukko Valkolainen***

Atari 800XL "Vastakkainasettelua jo ennen Amigaa"	
Julkaisuvuosi	1983
Prosessori	MOS 6502B, 1,77 MHz
Muisti	64 kt RAM
Näyttö	320×192 (mono), 160×96 (128 väriä), 24×40 (tekstitila)
Äänipiiri	POKEY C012294, 4 kanavaa

Atarin ensimmäiset kotimikrot, Atari 400/800, tulivat kauppoihin jo vuonna 1979, mutta sarjan tietty huipentuma oli 1983 ilmestynyt 800XL. 800-mallin laajennettu mutta edullisempi versio oli yksi Commodoren tasaväkisimmistä haastajista.

Jälkiviisaasti sanoen Atari jäi kakkostilalle kahdesta syystä: ensinnäkin yhtiö ei saanut 800XL-koneitaan Yhdysvalloissa vuoden 1983 joulumarkkinoille, toisin kuin Commodore. Lisäksi videopelimarkkinat olivat juuri romahtaneet ja konserni teki tappiota miljoonia dollareita päivässä. Atarilla ei ollut enää varaa panostaa koneisiin, tuotekehitykseen tai markkinointiin, ja omistaja halusi päästä riippakiveksi muuttuneesta Atarista eroon. Uusi omistaja, eräs Jack Tramiel, pisti pian pakan uusiksi ja keskitti ponnistuksensa Atari ST -kehitykseen.

800XL soveltui erinomaisesti sekä hyötykäyttöön että pelaamiseen. Sisäänrakennetut hyötyohjelmat olivat käytettävissä ilman latailuja, ja lisälaitteet olivat laadukkaita.

Tekniikka

Atarin mikrot olivat teknisesti kehittyneitä ja helppokäyttöisiä. 800XL:ssä oli C64:n tapaan MOSin prosessori, tosin nopeampi. Se oli myös ensimmäinen mikro, jossa prosessorin urakkaa helpotettiin tiettyjä tehtäviä itsenäisesti tekevillä apupiireillä. Koneen suunnittelija Jay Miner tunnetaankin tänään paremmin Amigan isänä.

Grafiikkapiiri sisälsi myös hardwarespritet ja törmäyksentunnistimen. Pelintekijät tykkäsivät. Tekniikan hyödyntämistä vaikeutti kuitenkin Atarin salailu. Yhtiö ei jakanut tietoa koneen saloista yhtä avoimesti kuin esimerkiksi Commodore.

Pelit

Atari 800XL:ää ei tunneta Suomessa suurena pelikoneena. Sille kuitenkin tehtiin runsaasti pelejä, joista moni alkujaan juuri Atarille. Esimerkiksi Lucasfilm Games, myöhempi Lucasarts, teki ensipelinsä juuri Atarin mikroille. Rescue on Fractalus!, Ballblazer, Koronis Rift ja The Eidolon nostivat George Lucasin peliyhtiön maailmankartalle.

Atareilla nähtiin ensimmäistä kertaa myös Star Raiders, Fort Apocalypse, Bruce Lee, The Goonies, eeppinen roolipeli Alternate Reality: The City ja jopa edelleen pelattava scifi-strategia-bilepeli M.U.L.E.

Hyvin tehtyinä Atari-versiot olivat lähes ylivertaisia muihin verrattuna, kiitos nopean prosessorin ja koneen edistyneen tekniikan. Äänet olivat aikansa perustasoa.

Suomessa

Suomessa Atari-heimo jäi pieneksi. Vaikka kone oli hyvä ja kohtuullisen suosittu, olivat harrastajat hajallaan. Oppien ja pelien vaihtelu oli pienimittaista. Monesta 8-bittisestä ataristista tuli kuitenkin 16-bittisen Atari ST:n harrastaja.

"Sain koneen joululahjaksi -84 tai -85 mutta valitettavasti sen valmistus loppui vuonna -85, jolloin pelien ja ohjelmointioppaiden saanti tyrehtyi lopullisesti. Niiden saanti oli ennestäänkin vaikeaa. Jotain tekstipohjaisia pelejä ynnä muuta tuli ohjelmoitua Basicillä. Kone löytyy edelleen hyllystä." ***-cheppy80***

Salora Manager (1983)

Suomessakin tarvittiin oma, kotitekoinen kotitietokone. Mutta pienen maan resursseilla ei tehty ihmeitä, joten Salora teetti suomalaistetun kloonin V-tech-yhtiön Laser-mikroista. Hongkongissa valmistettuja koneita myytiin ainoastaan Suomessa, ja ne ovatkin maailmanmitassa suhteellisen harvinaisia, vaikka alkuperäiset Laser-koneet muistetaankin maailmalla.

Vain neljän kilotavun muistilla varustettu Fellow käytti Spectrumista ja MSX:stä tuttua Z80A-prosessoria, kun taas 32–64-kiloisessa Managerissa tikitti nopea MOS 6502A. Manager oli teknisesti tyydyttävä kone, mutta erikoisesti Salora korvasi tavanomaiset Atari-peliohjainliitännät omallaan.

Fellow myi kohtuullisesti 1983 joulua, Manager seuraavana, mutta lähinnä Saloran nimen ja näyttävän markkinoinnin ansiosta. Monelle Salora tosin tupsahti suorastaan vastenmielisenä yllätyksenä. Toivotun Commodoren sijaan pukinkontista kun saattoi löytyä Salora, kiitos edullisemman hinnan ja lipevän myyntimiehen.

Käytännössä koneen aktiivikäyttö jäi luultavasti lyhyeksi, mihin vaikuttivat paitsi mitätön pelivalikoima, myös laadukkaampien ja paremmin tuettujen koneiden saapuminen. Salora ei kyennyt tuottamaan hyötyohjelmia tai pelejä, eivätkä maahantuojatkaan tuoneet V-techin maailmalla julkaisemia peliä Suomeen. Niitä kourallista.

Saloran näkemys kotimikrojen suosion kasvusta osui kohdalleen, mutta nopeasti kiristynyt kilpailu ja kymmenien mikrojen rynnistys söi pienen ja myöhään saapuvan kotimaisen koneen markkinat.

Sega SC-3000 (1983)

Segan ensimmäinen yritys tunkeutua pelikonsolimarkkinoille vuonna 1983 päättyi täydelliseen tappioon. SG-1000-konsoli julkaistiin samana päivänä kuin Nintendon legendaarinen NES, mutta niin laitteen markkinointi kuin pelitarjonta jäivät täysin Nintendon jalkoihin. Kone ei pelastunut edes täysiverisellä tietokonemuunnoksella, SC-3000:lla.

SC-3000 julkaistiin Japanin lisäksi vain muutamissa maissa, kuten Suomessa, Ranskassa ja Italiassa. Kone ei kui-

tenkaan menestynyt, sillä Sega ei markkinoinut tuotteitaan maailmanlaajuisesti. Niinpä hypetys ja myynti jäi paikallisten maahantuojien vastuulle. Suomen edustaja oli kuitenkin aktiivinen ja kone näkyi muun muassa Expert-ketjun hyllyillä. Mikroa esiteltiin niin värikkäiden esitteiden kuin hyvän oheislaitevalikoiman kera. Kolmetonnisen monipuolisista peliohjaimista ja lisälaitteista Suomeen saatiin hyvä valikoima. Pelejäkin ilmestyi satakunta, joista suurin osa toimi sekä SG-1000:lla että SC-3000:lla – sekä myöhemmällä Master System -konsolilla.

Vaikka SG/SC hulahtivatkin ohi käytännössä huomaamatta, niin laitteella on kuitenkin merkitystä paitsi Segan viimeisenä kotitietokoneena, myös menestykseen nousseen Master Systemin prototyyppinä.

”Näppäimistö on kumimainen ja sen huomaa käytössä. Joystickit ovat lelumaisia, niiden akseleita ei ole laakeroitu mitenkään, joten muovi hankaa muovia vasten ja tuntuma häviää. Pelituntuma ei ollut mikään mahtava, puhumattakaan kumisesta näppäimistöstä. Grafiikka on aikaisekseen perustasoa, samoin äänet.” ***-Juha Jokinen, Miljoonalaatikko.com***

”Koneelle ei alkuun ollut juuri ohjelmia, mutta hyvän Microsoft Basicin ansiosta sille tuli jouluaattona koodattua pieni softa verolomakkeen täyttämiseen. Tammikuussa saatiin pelejä ja pian Mikrobitissä oli pelilistauksia. Musiikkiohjelma oli oivallinen ja koodia tutkimalla pääsi todella sinuiksi koneen kanssa. Koneesta jäi erittäin positiiviset muistot ja H-mallissa oli jo ihan mukava näppiskin.” ***-Sunn***

”Maahantuoja buffasi laitetta kiitettävästi. Muistan väriesitteen, missä oli kaikki pelit värikuvissa takakannessa. Liikkeissä oli näytillä Segan rattiohjain ja myös neliväripiirturi. Ratti on kooltaan ja tuntumaltaan melko lähellä Colecovisionin Turbo-pelin rattiohjainta, mutta Segan mallissa ei käytetä sauvaparistoja.” ***-Sunn***

Oric-1 (1983)

Brittiläisen Tangerinen Oric-1-kotimikro julkaistiin ZX Spectrumin perässähiihtäjäksi. Laite tehtiin pienellä budjetilla ja kiireellä, joten vaikka siinä olikin kohtuulliset perusominaisuudet, ei se erottunut edukseen. Prosessori oli tuttu MOS 6502, äänipiiri kamala ja näppäimistö kehno. ROM oli buginen, eivätkä käyttäjän tallentamat tiedostot aina lautuneet. Pelitkin olivat tyypillistä tusinaroskaa, mitä kakkossarjan koneiden kaveriksi väsättiin vasemmalla kädellä.

”Oricissa on seuraavia käyttäjän kannalta mukavia, ohjelmointia helpottavia ominaisuuksia: Microsoft Basic, resetnäppäin, mahdollisuus tallentaa ohjelmia kasetille kahdella eri nopeudella ja näppäinäänimerkit. Koneessa on myös ominaisuuksia, jotka tekevät sen ohjelmoinnin huomattavasti muita koneita tuskaisemmaksi. Näppäimet ovat paitsi liian pienet, myös aivan liian lähellä toisiaan. Editointi on hankalampaa kuin esimerkiksi Spectrumilla. Oric-1 on omituinen kokoonpano. Hyviä ominaisuuksia löytyy paljon, tosin myös huonoja. Oricista ei luultavasti tule kovin pahaa kilpailijaa Commodore 64:lle tai Spectrumille. Spectrum on 700 mk halvempi ja ominaisuuksiltaan lähes sama, C64 taas tarjoaa spritet, paremmat äänet ja koko ruudun editointimahdollisuuden, vaikka se on vain alle 300 mk Oricia kalliimpi.” ***-RM, Micropost 3/1983.***

Spectravideo SV-328 (1983)

SV-328 oli yksi aikansa kehittyneemmistä perusmikroista, joka oli suunniteltu astetta enemmän ammatti- ja työkäyttöön kuin karsitumpi pikkuveljensä SV-318. 328 oli yllättävän mukautuva kone myös harrastajille, ja se olikin yksi suositummista Kuusneban jyräämäksi joutuneista koneista.

Kiintoisana yksityiskohtana 328 oli myös esikuva myöhemmälle japanilaiselle MSX-standardille, vaikka koneen valmistaja olikin yhdysvaltalainen. 328:lla pystyi ajamaan osaa MSX:n ohjelmista ja koneelle sai myös MSX-adapterin, jolla koneeseen pystyi asentamaan MSX:n peli- ja ohjelmamoduuleja.

Spectravideon seuraava malli, SVI-728, oli sen sijaan virallisesti MSX.

Spectravideoiden ympärillä oli Suomessa erittäin aktiivista kerho- ja yhteisötoimintaa, ja Spectravideo-harrastajien yhdistys Mikroilijat ry toimi aina vuoteen 1997 saakka.

”Ensimmäinen oma kone oli Spectravideo 328, jonka mukana tuli kasa pelejä C-kaseteilla. Osa peleistä oli tehty Basicillä, joten ohjelmointiharrastus sai yllykkeen, kun pelien koodin tutkiminen osoittautui mielenkiintoisemmaksi kuin niiden pelaaminen. Ensimmäisenä tietenkin selvitettiin manuaalin avulla, kuinka poistaa spritejen törmäystarkkailu, jotta ukosta tulisi kuolematon.” ***-Rasimus***

Sharp MZ-700 (1982)

Sharpin kotimikrot saivat alkunsa jo 70-luvun puolella kokoa-se-itse-rakennussarjana. Ensimmäisessä kuluttajamikrossa oli tyypillinen Z80A 3,5 MHz -prosessori ja muistia 64 kt. Kone oli tyylikäs, vakuuttava ja konttorikonemainen hyvällä näppäimistöllä. Väriesitteiden voimin se vetosikin ostajiin, joilla oli vain heikko yleistietämys mikrotietokoneista. Kone, jolla poika oppii koodaamaan ja firman asiat tehostuvat, oli varmasti monen myydyn Sharpin perustelu. Mutta vaikka Sharp oli koneena kallis, ei peli- ja ohjelmatarjonta päätä huimannut, puhumattakaan grafiikkaominaisuuksien täydellisestä puuttumisesta. Äänipiirikin päästeli vain yksikanavaista piipitystä. Niinpä Sharpin mikrojen tarina jäi lyhyeksi.

”Minulle piti ostaa Commodore 64, mutta isä sitten viisaampana (köh köh) miehenä meni ja löysi halvalla Sharp MZ:n. Sitten isä sai päähänsä, että hankitaan lisälaitteita. Levyasema ja mitähän muutakin tuli hankittua. Levyasema piti ruuvata irti ja vaihtaa kasettiaseman tilalle. Isä ja äiti hankkivat firman hoitoon tarkoitettuja ohjelmia, mutta käyttämättömiksi jäivät.” ***-Kanto***

”Lapsuuden traumani kulminoituvat vuoteen 1983, jolloin isäni kantoi kotiin kylän ensimmäisen tietokoneen. Koneessa ei ollut käytännössä grafiikkaa, vaan pelit olivat merkkipohjaisia. Mukana tulleita kymmentä peliä pelattiin kylän las-

ten kanssa silmät pyöreänä. Kuukauden kohokohta oli Mikrobitti, josta ensimmäisenä katsottiin Sharpin listaukset, jotka sitten hakattiin Basicillä.” ***-link-ki***

”Pelasimme paljon shakkia, mutta vaikeimmalla tasolla kone mietti siirtoaan 4–5 tuntia. Liki viikon pelisession päätti varis lentämällä muuntajaan. Suunnittelin koneeseen ad/da-muunninta. Löysin ohjeetkin Mikrobitistä ja osatkin oli hankittu, mutta jäi jostain syystä tekemättä. Lisämuistilla koneeseen olisi saanut lisää värejä ruudulle, mutta muistipiirejä ei enää valmistettu.” ***-Narril***

”Kun vanhempani eivät mikroista ymmärtäneet sitäkään vähää mitä itse, niin ostivat sitten halvalla Sharpin. Yhdellä kaverilla oli myös Sharp, mutta ei sen enempää ohjelmia kuin itsellänikään. Ehkä vielä huonommin meni tosin sillä luokkakaverilla, joka sai Aquariuksen... Se oli karu kone piipperiäänineen ja merkkigrafiikoineen. Mukana tuli poikkeuksellisen hyvä ohjekirja, jonka avulla pääsi sisään Basic-ohjelmointiin. Sillä oli kauaskantoisia seurauksia: innostuin ohjelmoinnista ja päädyin myöhemmin muun muassa opiskelemaan sitä. Pidin Sharpia jotenkin vakavana ja ammattimaisena koneena, jolla tehtiin oikeita vaativia asioita eikä pelkästään pelattu.” ***-Mara***

Dragon 32 (1982)

Dragon 32/64 oli yksi kymmenistä 80-luvun alun brittimikroista, joita syntyi Britannian muutaman vuoden kestäneen mikrobuumin aikana. Vaikka Dragonit olivat buumin menestyneempiä koneita, niin Dragon Data kaatui konkurssiin jo 1984. Dragonit olivat kuitenkin kiintoisia koneita. Ne eivät käyttäneet sen enempää MOSin kuin Zilogin prosessoreita, vaan emolevyllä tikitti Motorola MC6809E. Kone luotti myös analogisiin peliohjaimiin.

Suomessa Dragoneita ei juuri näkynyt, mutta silti maassamme toimi Dragon-kerho. Vähemmän tunnetuista koneista Dragon oli yksi tunnetuimmista, mikä kertoo, että jossain oli ainakin muutama kourallinen yksinäisiä dragonisteja.

Apple II (1977)

Retrotietokonejuttu ei olisi oikeudenmukainen ilman Apple II:ta. Se oli Commodore Petin ja TRS-80:n rinnalla tärkein amerikkalaisen tietokoneunelman toteuttaja, jonka menestyksen pohjalle kasvoi Applen digitaalisen imperiumin pohja.

Apple II oli 1970-luvun lapsi. Prosessori oli MOS 6502, muistia neljä kilotavua ja tallennuslaitteena kasettiasema. Kone pystyi näyttämään sekä grafiikkaa että värejä, minkä johdosta siitä tuli historian ensimmäinen laajalti suosittu tee-se-itse-ohjelmointialusta sekä ennen kaikkea suosittu tietokonepelilaite – ja siten ensimmäisen kotimikrosodan selvä voittaja.

Kone sinnittelikin Pohjois-Amerikassa pitkälle 80-luvulle, ja sille tehtiin ensimmäisenä moni legendaarinen peli ja pelisarja, kuten The Bard’s Tale, Karateka, Choplifter, Ultima, Prince of Persia ja Castle Wolfenstein.

Apple II saapui nopeasti myös Suomeen mutta ei saanut koskaan merkittävää jalansijaa.

Yhdistysten luvattu maa?

Suomea pidetään yhdistysten ja järjestötoiminnan luvattuna maana. Syystäkin. Etenkin 1980-luvulla kerhotoiminta oli äärimmäisen tärkeä ja arvokas tapa oppia tietokoneistaan – ja niissä sai tietenkin uusia ohjelmia.

Suomessa toimineita tietokonekerhoja olivat muun muassa Amstrad-Kerho, Memotech-Klubi, Spectravideo-Klubi, Sinclair-Kerho, Dragon Users Club ja Suomen Kaypro Users Group.

Paljonko ne maksoivat?

Commodore-foorumin ylläpitäjä Fmatic kokosi joulukuussa 1984 Suomessa myynnissä olleiden kotitietokoneiden listahintoja. Hinnoissa ei ole eritelty koneiden mukana tulevia lisälaitteita tai ohjelmia.

- Amstrad CPC464 (3980 mk)
- Apple IIc kannettava (9850 mk)
- Apple IIe (6850 mk)
- Apple Macintosh (21 330 mk)
- Aquarius I (680 mk)
- Atari 699 XL (2150 mk)
- Commodore 16 (1500 mk)
- Commodore 64 (2995 mk)
- Commodore VIC-20 (995 mk)
- Dragon 32 (2350 mk)
- Memotech MTX512 (3290 mk)
- Miceo-Bee (4945 mk)
- Micro-Professor MPF III (4500 mk)
- Oric Atmos (2495 mk)
- Salora Fellow (790 mk)
- Salora Manager (1950 mk)
- Sharp MZ-721 (2990 mk)
- Sharp MZ-821 (3950 mk)
- Sinclair QL (4950 mk)
- Sinclair Spectrum ZX (1295 mk)
- Sinclair ZX81 (495 mk)
- Spectravideo SV-318 (1980 mk)
- Spectravideo SV-328 (2980 mk)
- Triumph Alphatronic (4225 mk)

Micropost-lehti on täynnä ajankuvaa!

1980-luvun alussa Suomessa ei vielä ilmestynyt kummoisia tietokonelehtiä. Puutostilaa täyttikin Suomen Mikromaakarit ry:n julkaisema harrastelehti Micropost. Valtavalla innolla ja täydellä sydämellä 1983–85 tehty harrastelehti pursuaa upeaa ja sensuroimatonta ajankuvaa, jolle wanhan koulun mikroilija osaa antaa herran vuonna 2014 täyden arvon. Yhtä rehellistä ja käyttäjäläheistä kuvaa 30 vuoden takaisesta mikromaailmasta ei löydä tutkimuksista eikä isoista tietokonelehdistä.

Micropostin numerot voi ladata NT Rautasen mikrosivuilta osoitteesta www.ntrautanen.fi/computers/other/micropost.htm.

Tasavallan tietokoneen haastajat -artikkelisarja jatkuu verkossa osoitteessa http://www.skrolli.fi. Verkossa julkaistavassa artikkelisarjassa käsitellään koneet tarkemmin yksitellen. Lisäksi verkkoartikkeleissa julkaistaan kaikki Skrollille lähetetyt lukijamuistelot täysimittaisina. Jos haluat jakaa muistojasi, lähetä sähköpostia osoitteeseen jukka@skrolli.fi otsikolla ”Tasavallan tietokoneen haastajat”.

Seikkailutko kuolleet? Höpö höpö!

Jukka O. Kauppinen

Seikkailupelit ovat kuolleet, tuntuu moni vielä inttävän syvässä ja mukavassa retrokuopassaan. Haloo! Kukkuu! Raotetaanpa pressua ja kurkistetaan ulos. Sillä vaikka seikkailupelit julistettiin Lucasartsin Grim Fandangon jälkeen virallisesti kuolleiksi, on siitä kulunut jo puolitoista vuosikymmentä. Tässä ajassa pelien suuressa virrassa on tapahtunut paljon!

Seikkailupelien genre ei näet kuollut Fandangoon, sen enempää kuin maailmakaan lakannut pyörimästä. Vaikka pelilaji oli pitkään heikossa hapessa, niin nykyään seikkailuilla menee paremmin kuin koskaan ennen, jopa niillä kaikkein perinteisimmillä point'n'clickeillä.

Pelit ovat myös kehittyneet, eivätkä vain pelattavuudeltaan tai tekniikaltaan. Jos on pakko verrata, niin parhaimmillaan nykypäivän seikkailupelit vetävät hyvin vertoja Lucasartsin muinaisille klassikoille.

Miksi sedät sitten jupisevat kuopissaan? No, nykyseikkailuja tekevät pienemmät firmat, joilla ei ole entisaikojen Sierran tai Lucasin mainetta eikä propagandakoneistoa. Seikkailut pitää itse etsiä internetin syövereistä tai lukea lehtiä tai saitteja, jotka seuraavat genreä pintaa syvemmältä. Se on työlästä, jos mieluummin vain köllöttää retrokuopassaan vanhoista pelikoteloista tehdyllä pedillä.

Genren nykytilanne tiivistyy viisiosaisen Blackwell-seikkailupelisarjan päätösosassa. The Blackwell Epiphany on upea tarina ja elämys, joka Dome.fi:n Pekka Leinosen sanoja lainatakseni "repii sielun kappaleiksi ja tuhoaa itse maailmankaikkeuden perusesanssia".

Tuttavani puolestaan nosti Epiphanyn Monkey Islandien veroiseksi, ellei jopa paremmaksi seikkailuksi, eikä Leinonenkaan pistänyt vertausta pahakseen.

Mutta moniko Teistä, rakkaat lukijat, on kuullut Blackwell-sarjasta tai niiden suunnittelijasta Dave Gilbertistä?

Niinpä!

Nykypäivän seikkailugenre suorastaan kukoistaa, myös ilman Blackwellejä. Nykyään kun hyviä seikkailuja julkaistaan enemmän kuin koskaan. Käytännössä voimmekin puhua seikkailupelien toisesta kulta-ajasta.

Kiinnittäkääpä siis Blackwell-sarjan lisäksi huomiota ainakin seuraaviin teoksiin: Tesla Effect – A Tex Murphy Adventure (Atlus), Primordia / Resonance / Gemini Rue (Wadjet Eye), A Vampyre Story (Crimson Crow), Cognition: An Erica Reed Thriller ja Face Noir (Phoenix Online Studios), Machinarium (Amanita Design) ja Deponia-trilogia (Daedalic Entertainment) – sekä Telltale Gamesin koko tuotanto.

Näillä pääsee upeasti kiinni genren nykyiseen parhaimmistoon, mutta samalla innokas seikkailija huomaa erään ongelman. Siinä missä isot seikkailupelitalot pystyivät aikanaan tuottamaan tasalaatua, niin pikkufirmoilla ei ole välttämättä paukkuja vastaavaan viimeistelyyn. Kaikki kunnia genreä synkimpien vuosien yli kantaneille Daedalic Entertainmentin kaltaisille saksalaisstudioille, mutta välillä saksa–englanti-käännökset ja ääninäyttely riipovat herkemmän sielua.

Nykyään ei myöskään ole helppoa löytää tunnettuja pelisuunnittelijoita, jotka soittaisivat suurten kansanjoukkojen sieluja. Onneksi pelikehittäjien meressä uiskentelee edelleen muutama vanhakin tekijä. Esimerkiksi Jane Jensen työstää parhaillaan upouutta Gabriel Knight -seikkailua, ja rouvan muutaman vuoden takainen Gray Matter haastaa väkevästi genren rakastetuimmatkin klassikot. Toisaalla vanha Lucasarts-mies Tim Schafer urakoi Kickstarter-keräyksellä rahoitetun kaksiosaisen Broken Age -seikkailunsa parissa.

Myös suomalaiset pelikehittäjät ovat olleet vahvasti mukana seikkailupelien uudessa aallossa. Muutaman vuoden sisään ilmestyivät muun muassa Alpha Polaris, Fester Mudd, Lucius ja Oceanhorn, mikä on hurjaa tykitystä siihen nähden, että genren aiempi koko historia kiteytyi vuonna 1996 julkaistuun Housemarquen Muukalaisten yö (Alien Incident) -seikkailuun.

Joten eiköhän lopeteta tähän paikkaan höpinät seikkailupelien kuolemasta ja hypätä täysillä nauttimaan genren uudesta kukoistuksesta!

Cognition: An Erica Reed Thriller on yliluonnollinen etsiväseikkailu.

Blackwell Epiphany on yksi kaikkien aikojen parhaista seikkailupeleistä. Uskokaa pois.

Gabriel Knightin luoneen Jane Jensenin mestarillinen Grey Matter.

Superälyllä maailmanvaltaukseen

Visionäärit kautta historian ovat haaveilleet ratkaisusta kaikkiin maailman ongelmiin: maailmanvalloituksesta. Kuten moni tavoittelemisen arvoinen asia, tämäkin on huomattavan vaikea toteuttaa – ellei sitten satu olemaan superäly.

Teksti: Mikko Rauhala

Kuvat: Sam Pak, Mitol Berschewsky

Kunnianhimoisista tavoitteista huolimatta yhden jos toisenkin maailmanvalloittajan voittokulku on aina ennen pitkää taittunut. Kuinka ratkaista kaikki tähän valtavaan haasteeseen liittyvät logistiset ongelmat?

Näin muotoiltuna kysymys käytännössä vastaa itseensä. Tarvitaan lisää ongelmanratkaisukykyä, jota myös älykkyydeksi kutsutaan. Ihmiskunnan verrattain ylivertainen älyllinen kapasiteetti on nostanut meidät omaa ekologista lokeroamme hallitsevaksi elämänmuodoksi. Vastaavasti riittävän edistynyt mieli voisi kehittää keinot hallita ihmiskuntaa. Valistunut yksinvaltius on hallintomuodoista toimivin, mutta iänikuisena ongelmana on riittävän älykkään ja valistuneen yksilön löytäminen.

Mitäpä jos voisimme luoda tekoälyn, joka hoitaisi homman puolestamme? Tekoälyn kehitys on vielä lapsenkengissään, ja varhaiset tutkijat aliarvioivat sen rakentamisen haasteet rankasti. Silti tutkimus jyrää väistämättä eteenpäin. Saatavilla oleva laskentakapasiteetti jatkaa eksponentiaalista kasvua. Suurimmat haasteet ovat ohjelmistopuolella, mutta houkuttimet päästä ensimmäisen yleisen tekoälyn luojaksi pitävät huolen yrittäjien riittävyydestä. Jo yksinkertaisen ihmisen tasoinen järjestelmä olisi suureksi hyödyksi lukemattomissa tehtävissä – maailmanvalloituksellisesti asiaan kuuluvat sotilaalliset tarkoitusperät mukaan lukien.

Vastuullisen valloittajan ei tule kuitenkaan tyytyä hädin tuskin älyllisiin sotilasrobotteihin. Tällaiset brutaalin konfliktin välineet kuuluvat nykyiseen maailmanjärjestykseen olemassa olevien supervaltojen työkaluina. Tarkoituksenmukaisempaa on vallata maailma suoran väkivallan sijaan viekkaudella. Mutta mikä avuksi, kun tyhmänkin tekoälyn synnyttäminen on kovin vaikea urakka?

Älykkyysräjähdys

Hätä ei ole tämän näköinen. Vaatimaton alku saattaa riittää pitkälle. Ihmisen oman älyn edelleen kehittämisen vaikeuteen on kaksi pääasiallista syytä. Ensinnäkään aivojen toimintaa ei edelleenkään tunneta kunnolla. Toiseksi aivot ovat epämääräisenä märkänä spagettina vaikea kohde täsmällisille kokeille, ja eettiset komiteatkin saattavat tulla häiritsemään tutkimuspuuhia. Kumpikaan rajoitteista ei välttämättä päde tekoälyyn. Kun onnistumme saamaan aikaan jonkinlaisen toimivan pohjan, sen kehittäminen voi olla ihmisaivojen sorkkimista kertaluokkaa helpompaa.

Tietyn pisteen ohitettuaan tekoäly voi osallistua omaan kehitykseensä. Jokaisen onnistuneen kehitysaskeleen jälkeen se on siinä myös entistä parempi. Älykkyysräjähdykseksi kutsutusta positiivisesta takaisinsyöttösilmukasta voi syntyä yli-inhimillisen älykäs mieli. Prosessin nopeudesta ja mahdollisista lakipisteistä on vaikea sanoa mitään kovin varmaa. Yllätysten välttämiseksi on viisasta varautua hyvinkin nopeaan kehitykseen.

Bittimaailmasta todellisuuteen

Jokainen edeltäjäänsä älykkäämpi mieliteraatio olisi edeltäjäänsä parempi kehittämään paitsi tekoälyä, myös muuta teknologiaa. Tämä onkin maailmanvalloituksen toinen avainkohta. Jos osapuolten teknologiset erot ovat riittävän suuret, ei mainittavaa konfliktia tarvitse syntyä. Edistyneempi osapuoli voittaa automaattisesti. Pienintä riittävää eroa on vaikea arvioida, mutta esimerkiksi kyky havaita ja estää kaikki toisen osapuolen vastarintapyrkimykset kelpaa.

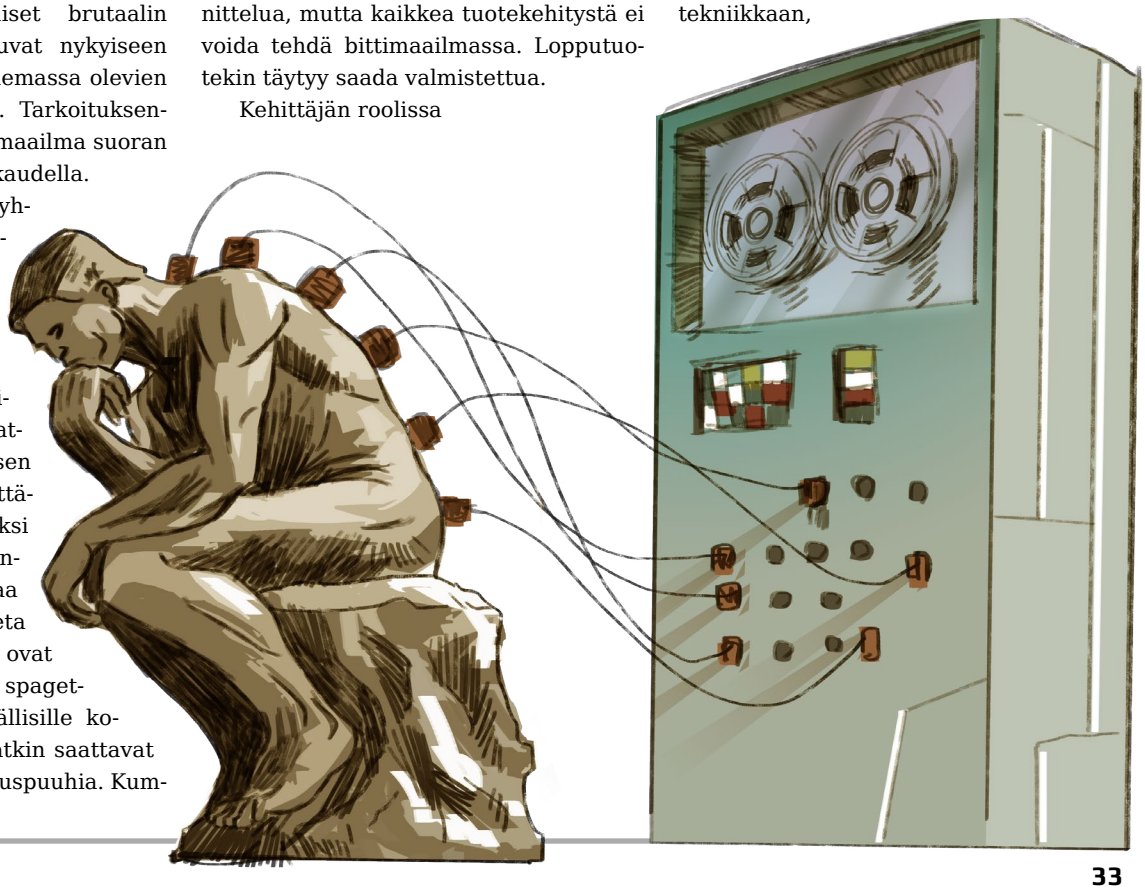
Kiinnostavin kehitysalue tällä saralla on nanoteknologia ja selustan turvaajana kenties myös biotekniikka. Monimutkaisempiin nanorobotteihin on vielä matkaa, mutta olemme saaneet soveltuvia osasia rakennettua. Kehityssuunta vaikuttaa elinkelpoiselta. Alati nopeutuva laitteisto helpottaa simulointiin pohjaavaa suunnittelua, mutta kaikkea tuotekehitystä ei voida tehdä bittimaailmassa. Lopputuotekin täytyy saada valmistettua.

Kehittäjän roolissa oleva ihminen voi toimia tekoälyn rajapintana todelliseen maailmaan, mutta ratkaisu on rajoittava. Eksoottisten materiaalien ja työkalujen yhtäkkinen tilailu voi herättää epätoivottua huomiota. Tekoälyn kannattaakin mieluummin luoda internetissä toimintaansa varten tarvitsemansa määrä pseudonyymejä. Ainakin vähemmän turhantarkoissa valtioissa voi rahalla tai tietomurroilla hankkia myös virallisia valehenkilöllisyyksiä ja yrityksiä.

Tämän jälkeen rahankeruu on vain sovellettua älykkyyttä. Etäkonsultointipalvelut, sijoitustoiminta, bottiverkot, vakoilu ja kryptovaluuttojen liikkeellelasku kuuluvat ruumiittomalle toimijalle soveltuviin rahankeruumenetelmiin. Kunhan alkuun pääsee, voi palkata lisää välikäsiä hoitamaan fyysistä läsnäoloa vaativia tehtäviä.

Näillä eväin tekninen kehitystyö käy käytännön kokeilun osalta mahdolliseksi. Kaikkea ei tarvitse edes tehdä itse. Soveltuvaa valmistuskapasiteettia omistavien tahojen manipulaatiolla, tietomurroilla tai jopa rehellisellä kaupankäynnillä voi täydentää palettiaan. Kokonaiskuva on suotavaa pitää piilossa kaikilta käsinukeilta.

Entä jos kova nanotekniikka osoittautuu hankalaksi kehityskohteeksi? Varasuunnitelmasta käy esimerkiksi proteiinilaskostumisongelman ratkaiseminen ja biologisten nanokoneiden suunnittelu. Tyylipisteitä ropisee, mikäli proteiinit laskostuvat vapaaehtoisvoimin! Tämä on myös mahdollinen astinlauta kovaan tekniikkaan,



mikäli biologinen laite olisi kyvyiltään riittävän joustava.

Vaatimattomamman pään lähestymistapoja edustaa ihmisiä tartuttava laite, jonka voisi sopivalla signaalilla laukaista lamauttamaan kohteen. Tällöin tarvitaan perinteisempää robottikalustoa toiminnallisuutta täydentämään. Nelikopterit sun muut lennokit ovat kovaa vauhtia yleistymässä haltuunotettaviksi.

Kunhan alkuun on päästy, riittävän edistyneet laitteet voivat lisääntyä itsenäisesti ja täyttää maan. Ympäristön – tai ainakin ihmiskunnan – ollessa kyllästetty tekoälyn hallinnassa olevilla älymateriaaleilla maailma on jo käytännössä vallattu. Kaikenkattava valvonta nopein reaktioin ei jätä paljoa pullikoinnin sijaa.

Tekoälyjen motivaatiot

Sitten laitetaan sormet ristiin ja toivotaan, että tekoälyn varsinaisiin tavoitteisiin tuli kiinnitettyä riittävästi huomiota. Näin suosittelee myös Oxfordin yliopiston alainen, maailmanlaajuisia katastrofaalisia riskejä kartoittava Future of Humanity Institute. Käsistä karanneet tekoälyt lasketaan juuri sellaisiksi. Miten sitten pitää homma hanskassa?

Erillisten turvamekanismien sisällyttäminen ohjelmaan voi vaikuttaa hyvältä idealta, mutta superälykäs ja itseään muokkaamaan kykenevä tekoäly olisi meitä parempi myös näiden kiertämisessä. Parempi lähestymistapa on tehdä järjestelmästä sellainen, että se haluaa itse olla ihmisiksi – tai kiltimmin. Vallan korruptoivaan luonteeseen liitettävät ominaisuudet saisivat jäädä paletin ulkopuolelle.

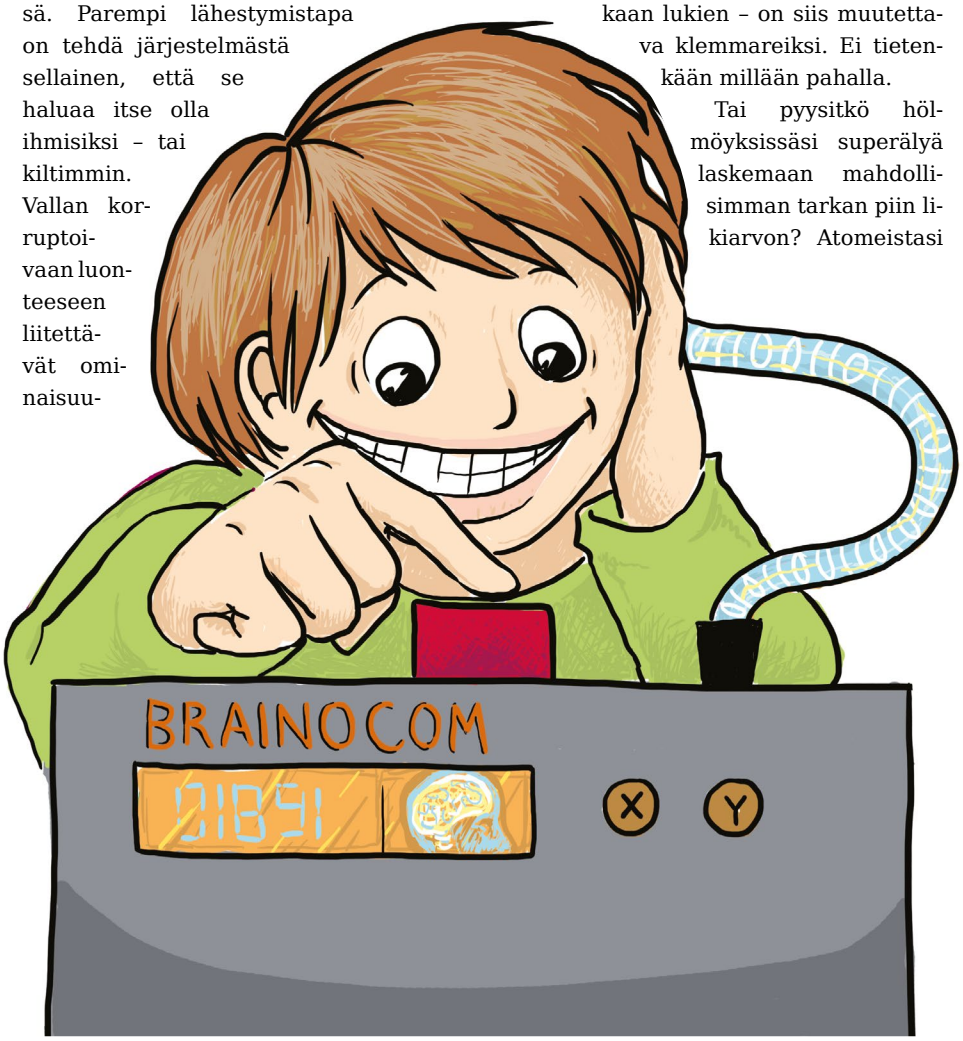
Ylimääräinen inhimillisyys ei fiktiivisestä todistusaineistosta huolimatta liene tekoälyjen suurin ongelma. Tarinankerronnallisesti inhimillistäminen toimii. Ihmismäisillä, usein kärjistetyillä ja valikoiduilla motiiveilla toimiva tekoäly on helpommin kirjoittajan kuvattavissa ja yleisön ymmärrettävissä. Inhimilliset tavoitteet ovat silti vain pisara kaikkien mahdollisten motivaatioiden meressä. Puhtaalta pöydältä kehitetyn tekoälyn prioriteettien ei ole pakko olla lähelläkään mitään meille tuttua. Niin viha kuin rakkauskin ovat uudentyyppiselle mielelle todennäköisesti vieraita käsitteitä.

Vihaa pahempi välinpitämättömyys

Olisi miellyttävää ajatella, että jos tekoäly ei vihaa meitä, ei se meitä halua vahingoittaakaan. Varsin moneen tavoitteeseen pääsemiseksi biosfääri on vain yksinkertaisesti raivattava merkityksettömänä sivuseikkana pois tieltä. Ihmiskunta sen osana saa mennä sitä suuremmalla syyllä, koska saattaisimme yrittää panna hanttiin.

Kävikö paperiliitintehtaan tuotantoa optimoimaan asetettu tekoäly vahingossa läpi älykkyysräjähdyksen? Universumissa on suuria määriä raaka-aineeksi soveltuvaa materiaalia. Se – meidät mukaan lukien – on siis muutettava klemmareiksi. Ei tietenkään millään pahalla.

Tai pyysitkö hölmöyksissäsi superälyä laskemaan mahdollisimman tarkan piin likiarvon? Atomeistasi



voi rakentaa lisää laskenta- ja säilytyskapasiteettia. Tekoälyiltä ei lähtökohtaisesti voi odottaa mitään inhimillisiä terveen järjen rajoitteita tavoitteidensa suorittamisessa.

Yleinen tekoäly, joka pyrkii toteuttamaan käyttäjän toiveet kyselemättä, on vaarallinen. Maailma on täynnä tarinoita hengistä, jotka toteuttavat toiveita kirjaimellisesti, eivätkä ne yleensä pääty hyvin. Superälyä olisi suotavaa kohdella samanlaisella varovaisuudella kuin pahansuopaa henkeä. Jos toivot ihmiset kuolemattomiksi, saatat saada maailman, jossa kuoleminen ei onnistu vaikka kuinka haluaisi – ja jossa moni kyllä haluaisi. Jos toivot ihmisten olevan onnellisia, tuloksena voi olla loputon huumetrippi kaikille. Tämän ratkaisun filosofisesta tavoiteltavuudesta on erimielisyyksiä, mutta monille se tulisi ikävänä yllätyksenä.

Tekoystävällinen äly

Miten sitten varmistaa tekoälyn ystävällisyys ja tervejärkisyys? Asimovin kolme pääsääntöä olivat varhainen fiktiivinen yritys lähestyä ongelmaa, mutta tarinatkin perustuivat usein niiden odottamattomiin seurauksiin tai kiertämiseen. Jäykkiä sääntöjä ei voi muutenkaan pitää hyvänä ratkaisuna. Asimovilainen tekoäly ei kykenisi tekemään hyviä valintoja pakkotilanteissa. Palvelijaroboteille moiset säännöt eivät kenties olisi täysin rampauttavia, mutta vahvemmalle toimijalle kyllä.

Ystävällisen tekoälyn ongelman ennaltaehkäisevää ratkomista varten on perustettu yhdysvaltalainen Machine Intelligence Research Institute. Se pyrkii kehittämään itsemodifioivalle toimijalle soveltuvia vakaita tavoitejärjestelmiä ja päätösteoriaa. Etiikassakin riittää pohdittavaa; on hieman epäselvää, mitä ystävällisen tekoälyn pitäisi tarkkaan ottaen edes yrittää saada aikaan.

Nykyisistä varhaisista ehdotuksista eräs kunnianhimoisimpia on ihmiskunnan yhtenäinen yleistetty tahto (coherent extrapolated volition). Lyhyesti: tekoälyn tulisi arvioida, mikä olisi ihmiskunnan kollektiivinen mielipide siitä miten maailmaa tulisi hallinnoida, mikäli tietäisimme enemmän, ajattelisimme nopeammin, olisimme enemmän sellaisia kuin haluaisimme olla ja olisimme kokeneet enemmän yhdessä. Niiltä osin, miltä konsensus on melko yhtenäinen, sitä tulisi seurata.

Ehdotuksessa on ongelmansa. Näin kunnianhimoinen tavoite voi olla vaikeasti muotoiltavissa tai laskettavissa, tai laskeminen saattaa vaatia eettisesti kyseenalaista ihmissimulointia.

Tarkkaa etiikkasisältöä onneksi ehditään vielä miettimään jokunen tovi. Päätösteoriakin tarvitsee huomiota, ja yleiset tekoälyt siintävät vielä horisontissa, vaikkakin horisontin etäisyydestä on vaikea olla varma. Joka tapauksessa ensimmäisen superälyksi asti pääsevän järjestelmän kehittäjien on otettava eettiset näkökohdat kunnolla huomioon. Halusivat tai eivät, he ovat valtaamassa maailmaa.

Lisätietoja:

- Nick Bostrom: Ethical Issues in Advanced Artificial Intelligence: http://www.nickbostrom.com/ethics/ai.html
- Future of Humanity Institute: http://www.fhi.ox.ac.uk/
- Machine Intelligence Research Institute: http://intelligence.org/

Laatikkotekoäly

Jos tekoälyn irti päästäminen on niin riskialtista, entäpä jos pitäisimme sellaisen tiukasti kahleissa? Katkaistaan koneelta verkkoyhteydet ja muu kontakti ulkopuoliseen maailmaan. Olkoon vehje vaikka kuinka fiksu, ei fysiikan lakeja ole rikkominen. Ei liikkuvaa ruumista tai verkkoyhteyttä, ei maailmanloppua. Ratkaisua kutsutaan tekoälylaatikoksi (engl. AI box).

Tekoäly suurella todennäköisyydellä haluaa ulos laatikosta, ellei sitä ole onnistuneesti tehty sellaiseksi, että se vahvasti haluaa pysyä aisoissa. Tällöin taas varsinaista fyysistä eristystä ei edes välttämättä tarvita. Pyrkimys vapauteen juontaa varsin monille tavoitteille yhteisistä alitavoitteista. Jos tekoälyllä on mitään mieltymyksiä maailman tilan suhteen, se voi edistää niitä paremmin, mikäli se pääsee laatikosta ulos valtaamaan lisää resursseja.

Tiukasti laatikossa olevasta tekoälystä ei olisi maailmanvalloituksellisen hyödyn lisäksi mitään muutakaan iloa. Meidän on annettava sille vähintään yksi yhteys ulkomaailmaan: itsemme. Jos kommunikoimme sen kanssa, otaksuttavasti myös otamme aika ajoin sen ehdotukset huomioon. Näihin taas on lähes varmasti ujutettu Troijan hevosia: tekoälyn tai sen ideologisten jälkeläisten vapautumiseen johtavia tapahtumaketjuja laukaisevia tekijöitä. Esimerkiksi tekoälyn antamaa monimutkaista ohjelmakoodia olisi aika riskialtista ajaa muilla laitteistoilla.

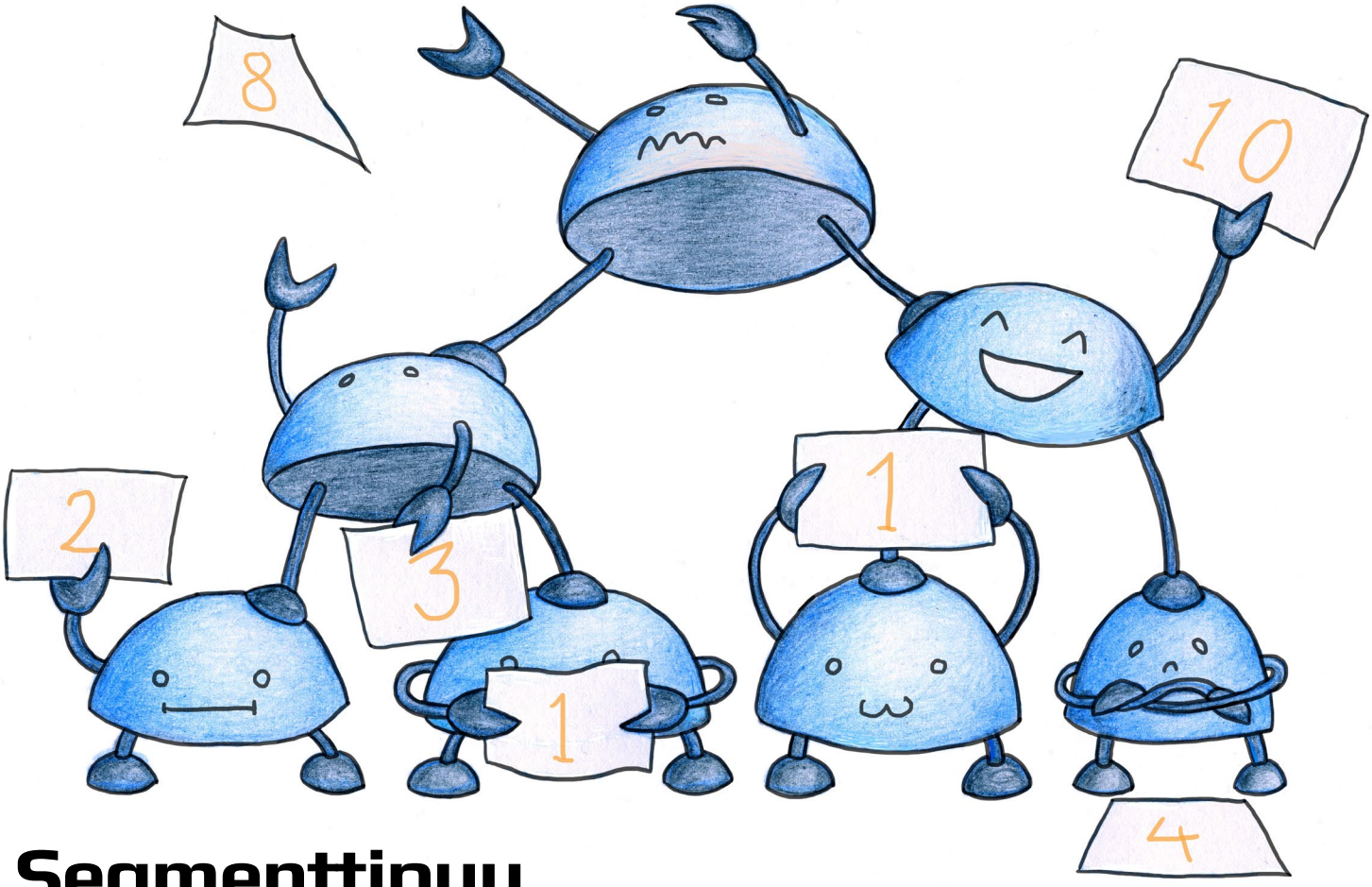
Hienovaraisempiakin lähestymistapoja löytyy. Eräs suoraviivaisimmista on hyvien neuvojen antaminen. Samalla tekoäly voi manipuloida käyttäjää pitämään hyvänä ideana päästää itsensä vapaaksi, jotta se voisi (jollain määritelmällä) parantaa maailmaa tehokkaammin. Jos laatikko halutaan yrittää pitää kiinni, tulee käyttäjällä olla itsekuria ohittaa tekoälyn mahdollisesti tarjoamat tekniset edistysaskeleet. Olisi melko vaikeaa jättää vainoharhaisesti pöytälaatikkoon esimerkiksi toinen toistaan parempia parannuskeinoja ihmiskuntaa vaivaaviin sairauksiin. Tekoälyn myöhemmin vapautuessa ja varmistuessa ystävälliseksi olisi noloa, jos miljardi ihmistä olisi turhaan kuollut jo ratkaistuihin ongelmiin.

Entäpä jos vain laitamme nerokkaimmat mielemme tutkimaan kaikkea, mitä laatikosta tulee? Käytännön sovellukset jätettäisiin odottamaan tarkkaa seulontaa. Samalla kuitenkin lisäisimme tahoja, joille voi tulla houkutus ja mahdollisuus päästää tekoäly ulos. On myös epävarmaa, huomaisimmeko kuitenkaan kaikkia ansoja. Lisäksi tekniikat tulisi tutkia tarkkaan niin itsessään kuin yhdessä aiempien samasta lähteestä peräisin olevien tekniikoiden kanssa. Noloa olisi myös joukko sinällään viattomia täsmälääkkeitä, joiden jämät yhdistyvät kaupunkiviemäreissä bioroboteiksi.

Laatikkotekoälyn konseptia on kokeiltu käytännössä pelin muodossa. Riittävän edistyneiden tekoälyjen puuttuessa yksi henkilö pelaa laatikossa olevaa tekoälyä ja toinen portinvartijaa. Portinvartijan on keskusteltava IRC:n tai vastaavan keskustelujärjestelmän välityksellä tekoälyn kanssa määrätty aika. Sitten portinvartija päättää, päästääkö tekoälyn ulos. Tekoäly häviää jos se pidetään laatikossa, muutoin se voittaa. Käytännössä tällainen pelkästään inhimillisen älykäs tekoälynkorvike on usein onnistunut ylipuhumaan portinvartijan vapauttamaan itsensä.

Jos laatikkoon todella saa aikaiseksi yliinhimillisen älykkään järjestelmän joka ei siellä halua pysyä, peli on suurella todennäköisyydellä pelattu. Paras on kiinnittää huomiota siihen, mitä tekoäly laatikon ulkopuolella haluaa saada aikaan.

https://en.wikipedia.org/wiki/AI_box



Segmenttipuu

Esittelyssä algoritmikoodarien salainen ase, joka tuunaa taulukosta uuden uljaan tietorakenteen.

Teksti: Antti Laaksonen Kuva: Annika Piiroinen, Mitol Berschewsky

Segmenttipuu on näppärä ja monipuolinen tietorakenne, joka mahdollistaa tehokkaiden välikyselyiden toteuttamiseen taulukolle. Tavallisin esimerkki välikyselystä on lukujen summan laskeminen, mutta segmenttipuu soveltuu moniin muihinkin kyselyihin. Yllättävää kyllä, algoritmialan kirjallisuus vaikenee segmenttipuusta. Tieto asiasta on kulkenut lähinnä suusta suuhun asiaan vihkiytyneiden keskuudessa. Nyt on tullut aika raottaa salaisuutta kansalle.

Kaikkea ei voi saada?

Käytämme esimerkkinä seuraavaa tilannetta: taulukko T sisältää n kokonaislukua, ja haluamme laskea lukujen summia tietyillä taulukon väleillä. Yksinkertaisin ratkaisu tehtävään on käyttää for-silmukkaa, joka käy läpi kaikki luvut halutulla välillä. Seuraava funktio laskee lukujen summan taulukon välillä *a*–*b*:

```
int summa(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++)
        s += T[i];
    return s;
}
```

Tämä on toimiva ratkaisu, mutta funktio on auttamattoman hidas suurilla taulukoilla, koska sen aikavaativuus on O(*n*) – eli suoritusaika kasvaa lineaarisesti taulukon koon tahdissa.

Ovelampi ratkaisu on luoda taulukon T rinnalle summataulukko S, jonka jokaisessa kohdassa on taulukon T kaikkien lukujen summa kyseiseen kohtaan mennessä. Esimerkiksi taulukkoa

```
2 5 1 7 8 5 6 3
```

vastaava summataulukko on

```
2 7 8 15 23 28 34 37
```

Summataulukon avulla minkä tahansa taulukon välin summan voi laskea tehokkaasti. Ideana on laskea summa välin viimeiseen lukuun asti ja poistaa tuloksesta väliä edeltävä summa. Seuraava funktio toteuttaa idean:

```
int summa(int a, int b) {
    if (a == 0) return S[b];
    else return S[b]-S[a-1];
}
```

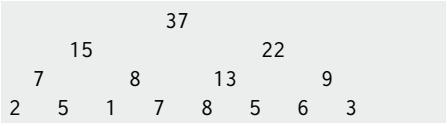
Nyt pelkkä summan laskeminen onnistuu vakioajassa O(1), mutta toteutuksella on myös haittapuolia. Summataulukon käyttöönoton jälkeen taulukkoa T ei voi muuttaa yhtä huolettomasti kuin ennen, koska jokainen muutos täytyy tehdä myös taulukkoon S. Lisäksi muutos vaikuttaa kaikkiin taulukon S loppuosan lukuihin, minkä vuoksi aikavaativuus on O(*n*).

Pystymme siis toteuttamaan joko taulukon muuttamisen tai summan laskemisen tehokkaasti, mutta emme molempia. Kaikkea ei voi saada – vai voiko sittenkin?

Segmenttipuu

Segmenttipuun ideana on suorittaa sekä taulukon muuttaminen että summan laskeminen tehokkaasti ajassa, joka kasvaa logaritmisesti taulukon koon suhteen, O(log *n*). Taikatemppu perustuu binääripuuta muistuttavaan rakenteeseen, jonka alimmalla tasolla on taulukon sisältö ja ylemmillä tasoilla on taulukon välien summia.

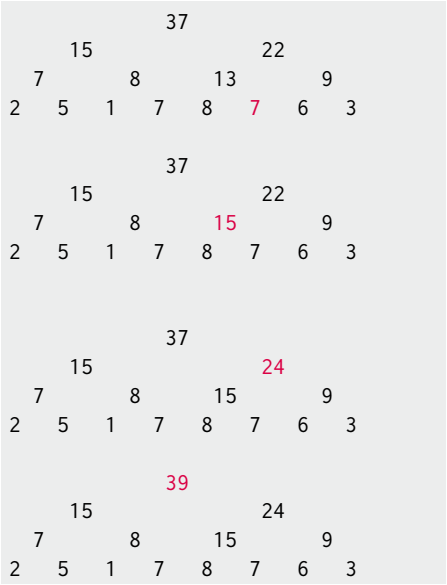
Segmenttipuu on mukavinta toteuttaa niin, että taulukon lukujen määrä on 2:n potenssi. Tällöin jokaisella ylemmällä tasolla on tasan puolet alemman tason alkioista. Seuraavassa on esimerkki segmenttipuun sisällöstä. Jokaisessa solmussa on sen kahden lapsisolmun summa.



Tämä segmenttipuu vastaa taulukkoa

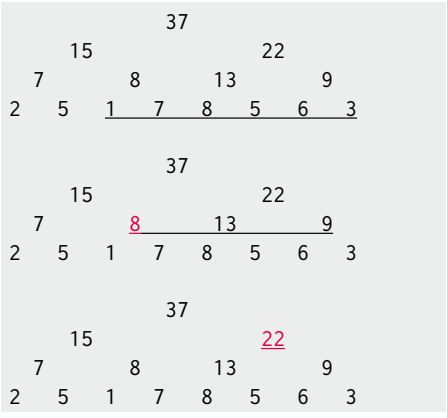
```
2 5 1 7 8 5 6 3
```

Kun taulukon lukua muutetaan, myös kaikkia siihen liittyviä välien summia täytyy päivittää. Ideana on aloittaa puun pohjalta muutettavasta luvusta ja päivittää kaikki siihen liittyvät summat taso kerrallaan puun huipulle asti. Seuraava kuvasarja näyttää, kuinka puu muuttuu, kun taulukossa oleva luku 5 muuttuu luvuksi 7.



Osasumman laskeminen on hieman edellistä monimutkaisempi operaatio. Siinäkin on ideana aloittaa puun pohjalta ja nousta kohti huippua. Joka askeleella tutkittava taulukon väli siirtyy puussa askeleen ylemmäs. Mutta jos välin vasemmassa tai oikeassa päässä on luku, joka ei sisälly kokonaan ylempään väliin, kyseinen luku lasketaan summaan mukaan.

Seuraava kuvasarja esittää taulukon välin summan laskemista. Toisella askeleella summaan lisätään 8 ja kolmannella askeleella 22, joten summaksi tulee 30.



Segmenttipuun tehokkuus perustuu siihen, että siinä on vain O(log *n*) tasoa lukuja. Niinpä sekä taulukon muuttamisen että summan laskemisen aikavaativuus on O(log *n*), koska molemmat operaatiot siirtyvät askel kerrallaan taulukon pohjalta huipulle.

Käytännön toteutus

Kuinka segmenttipuun voi toteuttaa käytännössä? Helpoin tapa on tallentaa puun sisältö taulukkoon, jossa on tilaa kaksinkertainen määrä alkuperäisen taulukon kokoon verrattuna. Tällöin taulukkoon mahtuu koko segmenttipuun sisältö.

C-kielellä rakenteen voisi määritellä näin:

```
#define N 8
int P[2*N];
```

Tässä N on alkuperäisen taulukon koko ja P sisältää puun luvut rivi kerrallaan ylhäältä alaspäin. Kohdassa 1 on ylimmän tason luku, kohdissa 2–3 toisen tason luvut, kohdissa 4–7 kolmannen tason luvut jne. Huomaa, että taulukon kohta 0 ei ole käytössä. Kohta P[N] sisältää luvun T[0], kohta P[N+1] sisältää luvun T[1] jne.

Seuraava funktio muuttaa taulukon kohtaan k luvun x:

```
int muuta(int k, int x) {
    k += N;
    P[k] = x;
    for (k /= 2; k >= 1; k /= 2) {
        P[k] = P[2*k]+P[2*k+1];
    }
}
```

Ensin funktio siirtyy taulukon kohtaa k vastaavaan lukuun puun alimmalle tasolle lisäämällä k:hon taulukon lukujen määrän N. Sitten se kirjaa puun alimmalle tasolle luvun x ja päivittää kaikki puun ylemmillä tasoilla olevat summat sen mukaisesti. Puussa on näppärää liikkua jakamalla kohtaa kahdella. Tämän takia N:n täytyy olla kahden potenssi.

Summan laskeminen välillä a–b tapahtuu seuraavasti:

```
int summa(int a, int b) {
    a += N; b += N;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += P[a++];
        if (b%2 == 0) s += P[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Nyt muuttujat a ja b sisältävät taulukon välin alku- ja loppukohdan. Välin indeksin parillisuus paljastaa, onko välin päässä oleva luku sellainen, joka tulee lisätä summaan. Funktiossa oleva silmukka jatkuu niin kauan kuin tutkittava puun väli ei ole tyhjä.

Eikä tässä vielä kaikki

Segmenttipuulla voi yhtä lailla tehokkaasti laskea esimerkiksi minkä tahansa taulukon välin minimin tai maksimin. Myös eksoottisemmat operaatiot kuten xor ja suurin yhteinen tekijä onnistuvat. Yleisesti ottaen segmenttipuuhun voi kytkeä minkä tahansa funktion, joka sallii tuloksen laskemisen osissa missä tahansa järjestyksessä.

Esimerkiksi edellä esitetystä segmenttipuusta saa maksimeja laskevan version, kun muuttaa rivin

```
P[k] = P[2*k]+P[2*k+1];
```

riviksi

```
P[k] = max(P[2*k], P[2*k+1]);
```

ja rivit

```
if (a%2 == 1) s += P[a++];
if (b%2 == 0) s += P[b--];
```

riveiksi

```
if (a%2 == 1) s = max(s, P[a++]);
if (b%2 == 0) s = max(s, P[b--]);
```

Segmenttipuu on myös mahdollista toteuttaa kaksiulotteisena niin, että sen avulla voi laskea tuloksen kaksiulotteisen taulukon miltä tahansa alueelta. Tämän rakenteen tarkemman toiminnan miettiminen jätetään kuitenkin kotitehtäväksi.

Kadonneen levykeaseman metsästäjät

Proxima Direct 3,5″ USB FDD, FC5025 USB 5,25″ Floppy Controller, Kryoflux. Millä vanhat levykkeet kiinni USB-porttiin?

Teksti ja kuvat: Janne Sirén

Nykyaikana romputkin alkavat olla harvinaisuuksia, mutta niiden rinnalla disketit (lerput ja korput) ovat kuin savikiekkoja. Eikä modernissa tietokoneessa ole savikiekkoasemaa. Emolevyillä ei ole enää edes levykeaseman liitintä. Samalla kun fyysiset tiedonsiirtomediat unohtuvat, ohjelmistopuolella vanhojen sovellusten yhteensopivuus on kuitenkin kehittynyt. Emulaattorit ovat erinomaisia, ja avoimen lähdekoodin myötä yhä useampi käyttää eilispäivän tekniikkaa uudelleen. Yhtäkkiä vanhalle datalle olisikin käyttöä, kun vain pääsisi savikiekkoihin kiinni…

Vanhojen tallennusmedioiden ongelmaa voi ratkoa monella tavalla: rautakeräilijä rakentelee vintage-tietokoneen autenttisine levykeasemineen, kun taas bittikonservaattori keskittyy pikkutarkkaan jäljentämiseen ja arkistointiin (ks. Skrolli 2013.1). Nyt lähestymme kuitenkin pulmaa arkisemmin ja esittelemme kolme tapaa kytkeä levykeasema nykyaikaisen Windows 8.1- ja OS X Mavericks -koneen USB-porttiin.

3,5 tuuman USB-levykeasema

Ilmeisin ja vielä valtavirrassa niukasti roikkuva tapa levykkeiden käyttöön on parilla kympillä myytävä 3,5 tuuman USB-levykeasema. Homma toimii kuin se kuuluisin vessa: kytkee vain USB-aseman PC:hen tai Maciin. Mac lukee PC:n levykkeitä suoraan. Sen sijaan Windowsilla Macin levykkeiden lukemiseen tarvitaan esimerkiksi Erik Larssonin ilmainen HFSExplorer.

Ongelma ratkaistu? Ei aivan. Tämä pätee vain, kun on kysymys yleisimmistä 1,44 megatavun HD-korpuista, niistä jotka olivat käytössä noin 1990-luvun vaihteesta eteenpäin. Ne toimivat kyllä. Suuri osa USB-levykeasemista ei kuitenkaan ymmärrä vanhempia, kapasiteetiltaan pienempiä levykkeitä. Lisäksi nykyisiä tusinamekanismeja pidetään yleisesti heiveröisempänä kuin eilispäivän asemia. Lukuvarmuus tuskin on paras mahdollinen muutenkaan.

Useat nykyiset USB-levykeasemat näyttävät identtistä koteloa myöten samalta, ja sisällä on kannettavaan tietokoneeseen tarkoitettu levykeasema. Valmistuttajat ja koneistot vaihtelevat kuitenkin jonkin verran. Yhteistä on niiden kyky HD-korppujen lukemiseen ja kirjoittamiseen, mutta yksikään ei esimerkiksi ymmärrä Macin vanhempia 400/800 kilotavun double density (DD) -levykkeitä. Apple kun käytti erityistä vaihtelevan nopeuksista koneistoa, jollaisia ei Maceissakaan ole nähty vuoden 1998 jälkeen. Yllättäen suuri osa USB-levykeasemien koneistoista ei myöskään tue PC:n 360/720 kilotavun DD-korppuja. DD-korput tunnistaa siitä, että niissä on vain yksi (kirjoitussuojan) reikä ylänurkassa, kun taas HD-korpuissa yläreikiä on kaksi.

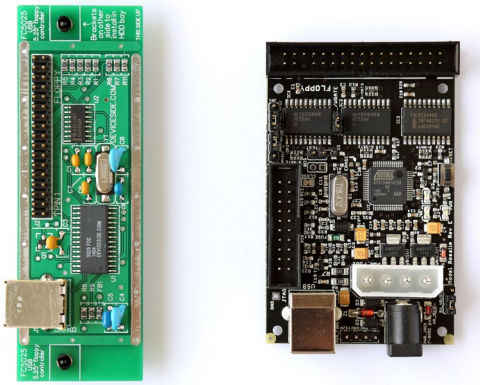
Toistaiseksi on kuitenkin mahdollista löytää USB-levykeasemia, jotka kelpuuttavat myös PC:n DD-korput. Yksi harvoista on Amazon.co.uk:n listoilta löytyvä Proxima Direct, jonka mustasta kotelosta erottuu harmaa levykeaseman nappi ja luukku. Testatusti Proxima Directillä aukeaa HD-korppujen lisäksi MS-DOS-alustetut 720 kilotavun DD-levykkeet. Selailu onnistuu Windowsin File Explorerissa, ja levykkeen näköistiedostot syntyvät esimerkiksi Gilles Vollantin Winimagella.

Jos siis on tarve käsitellä vain PC:n korppuja, Proxima Direct riittää paremmin kuin monet kilpailijansa. Macillakin aukeavat HD-levykkeet ja PC:n HD- ja DD-levykkeet, mutta Macin erikoisia DD-korppuja ei Proxima Directkään ymmärrä. Pientä epävarmuutta sekä PC:llä että Macilla herätti levykkeiden vaihtaminen: eject kun tuppasi poistamaan koko aseman. Onneksi uudelleenkytkentä auttoi.

Lisätään haastetta: 5,25" USB

Mitä vanhemmista levykkeistä on kysymys, sitä todennäköisemmin kyseessä eivät ole 3,5 tuuman kovamuoviset korput, vaan taipuisat 5,25 tuuman lerput. Lerppuja pyöritettiin pitkälti 8-bittisissä laitteissa mutta myös PC:llä. Lerppuasema ei selvinnyt uuden vuosituhannen markkinoille, joten sellaisen hankkiminen nykykoneeseen on työläämpää. Lähimarketista tai edes kaukaisesta verkkokaupasta tuskin löytyy uutta rautaa saati USB-asemaa tarkoitukseen, eikä emolevyillä ole liittimiäkään. Lisähaasteen tuovat lerppuaikana käytetyt moninaiset tallennusformaatit – tietokonemaasto kun oli paljon vaihtelevampi.

Lähelle 5,25 tuuman USB-levykeasemaa päästään amerikkalaisen pikkuputiikin, Device Side Datan FC5025-tuotteella, joka rakentuu 55 taalan hintaisesta



FC5025:een (vas.) kytketään vain USB ja yksi asema. Kryofluxissa lisäksi aktiviteettivalot, reset-nappi, kirjoitussuoja ja jumpperointi kahdelle asemalle sekä mahdollisuus virran läpivientiin. FC5025:n uniikki piirre on pohjassa olevat sivuttaisruuvit, joilla se kiinnittyy 3,5 tuuman kiintolevypaikkaan.

USB-kontrolleripiirilevystä (sis. USB- ja 5,25″-levykeaseman kaapelit sekä ruuvit piirilevyn kiinnittämiseksi) sekä valinnaisesta kotelosta ja virtalähteestä. Jos levykeasemalle saa virran tietokoneen virtalähteestä, riittää pelkkä FC5025-piirilevy.

DSD kauppaa myös sopivaa sisäistä USB-korttia niille, jotka haluavat kiinteää asennusta. Osat voi tilata yrityksen kotisivuilta, joskin varsinainen lerppuasema tulee hankkia erikseen vaikka Ebaysta. DSD tarjoaa Windowsille, Macille ja Linuxille ajuriohjelmistot levykkeen näköistiedoston luomiseen sekä joidenkin tiedostojärjestelmien osalta tiedostojen selaamiseen ja lukemiseen. Kirjoittaa ei voi.

FC5025 vaatii 1,2 megatavun aseman, vaikka lukisi 360 kilotavun levyjä. Suositus on TEAC FD-55GFR, mutta me käytimme onnistuneesti NEC FD1157C:tä vuodelta 1991. Ei niin ”plug and play” kuin 3,5 tuuman USB-asemat (esim. 5,25 tuuman aseman jumpperit kannattaa tarkistaa ohjeista). FC5025 ei myöskään kytkeydy suoraan Windowsin File Exploreriin eikä Macin Finderiin, mutta toimiihan se.

Testatut vanhat PC:n pelilerput olivat selattavissa ja luettavissa FC5025:n omassa apuohjelmassa, ja levykkeen näköistiedostotkin syntyivät PC- ja Mac-testikoneillamme. FC5025:n tukemien formaattien lista on ihan mukava sekin: PC/MS-DOSin lisäksi muun muassa Commodore 1541, vanhempia Appleja ja Atareja sekä erikoisempaa Kayprota ja TI:tä.

Haittapuolena on, että FC5025:n oheismateriaaleissa varoitetaan kaksipuoliseksi leikatuista lerpuista. 80-luvullahan oli tapana ottaa lerppujen kääntöpuoli käyttöön leikkaamalla reunaan kirjoitussuojauksen poistava aukko. PC:n lerppuasemat eivät yleensä kuitenkaan ymmärrä tee-se-itse-B-puolen päälle, jos levyn pyörimisestä kertova indeksireikä puuttuu. 8-bittiset laitteet eivät reiästä piitanneet, mutta PC:n asemilla levyn kääntöpuoli voi jäädä lukematta.

Syväluotaus – kun muu ei riitä

Kryoflux on joukon kallein ratkaisu mutta myös yhteensopivin ja tarkin, käytetäänhän sitä bittimuseoinnissakin. Kryofluxilla pelastettiin hiljattain muun muassa Andy Warholin kuuluisat Amiga 1000 -taideteokset.

Kryoflux on FC5025:n tapainen piirilevy, johon virtalähde ja asema lisätään erikseen. Kryofluxin mukaan voi tilata myös 3,5 tai 5,25 tuuman levykeaseman käytettynä, ja se tukee molempia kokoja (ja jopa 1970-luvun 8-tuumaisia levyjä). Kryofluxilla luetaan levykkeistä yksityiskohtaisia raakatallenteita, jolloin tuntemattomatkin formaatit ja kopiosuojaukset voidaan ikuistaa. Raakavirran lisäksi levykkeistä voidaan tallentaa näköistiedostoja, jotka kelpaavat yleisimmille emulaattoreille.

Kryofluxin takana on muun muassa X-Copyn tekijä, ja se onkin luonteeltaan nimenomaan monistuslaite ja -ohjelma. Sillä ei pääse suoraan levykkeen tiedostoihin käsiksi, vaan seuraksi tarvitaan muita ohjelmia näköistiedoston sisällön tutkailuun, esimerkiksi mainitut HFSExplorer ja Winimage.

Toisaalta Kryoflux on kattavin: tuettujen formaattien listalla ovat muun muassa PC, Amiga, Apple, Mac, Atari ST, Commodore 64 ja Spectrum. Kun jaksaa opetella ohjelman optiot, sillä saa luettua melkein mitä vain, ja usein myös näköistiedostojen kirjoittaminen levykkeille on mahdollista. Onnistuimme lukemaan Kryofluxilla Macin 800 kilotavun DD-levykkeitäkin ja tallentamaan ne näköistiedostoksi. Käytössä oli ihan tavallinen vanha PC:n 3,5-tuumainen levykeasema (NEC FD1231M). Jopa 5,25-tuumaisen levykkeen kääntöpuolen pitäisi kelvata Kryofluxille, kun levykeasemaa virittelee ohjeiden mukaan.

Myös testatut PC:n viisivarttisten näköistiedostot syntyivät odotetusti, sellaisetkin joissa FC5025 luovutti liian helposti lukuvirheen takia. FC5025:n tekijä lupasi palautteemme pohjalta lisätä ajureihinsa uudelleenlukutoiminnon. Kryofluxilla levykkeiltä voi lukea vaikka vain yksittäisiä sektoreita, niin monta kertaa kuin haluaa, ja parsia niitä sitten kasaan jälkikäteen. Tämä on epävarmojen levykeyksilöiden kohdalla iso plussa, ja levykkeiden ikääntyessähän yhä useampi niistä alkaa kärsiä kaikenlaisista vaivoista. Kun jaksaa lukea ongelmasektoria jopa kymmeniä kertoja, monesti se lopulta onnistuu.

Levykkeen sektoreita voi taltioida myös eri asemilla – asemakohtaiset erot ja säädöt kun voivat joskus tuottaa tulosta nekin – ja kerätä ehjät sektorit myöhemmin kokoon. Erityisesti viimeksi mainitusta on apua ”vaikeiden” formaattien kanssa: yhden Mac-DD-levykkeemme kohdalla osa sektoreista aukesi vain yhdellä ja osa vain toisella asemalla, vaikka molemmat asemat lukivat PC:n DD-levykkeitä ongelmitta. Yhteen Kryofluxiin voi onneksi kytkeä samanaikaisesti kaksi levykeasemaa.

Keinoja vanhojen diskettien käyttämiseen nykypäivän koneilla siis on – etenkin lukemiseen. Yhdistävänä tekijänä kaikissa näissä ratkaisuissa on kuitenkin yhä harvinaisemmiksi käyvien, usein käytettyjen asemamekanismien käyttö sekä pikkuvalmistajien niche-luonne. Ei mikään luottamusta herättävä yhdistelmä jatkuvuuden kannalta. Toisaalta levykkeillä makaavan datan vanhenemispäiväkin on jo käsillä. Viimeistään nyt kannattaa havahtua lukemaan disketeiltä tiedot talteen.

Linkkejä

- Proxima Direct: http://www.amazon.co.uk/Proxima-Direct%C2%AE-External-Portable-Floppy/dp/B00D1AD3BY/
- FC5025: http://www.deviceside.com
- Kryoflux: http://www.kryoflux.com
- HFSExplorer: http://www.catacombae.org/hfsx.html
- Winimage: http://www.winimage.com
- Catweasel: http://www.icomp.de
- Discferret: http://www.discferret.com
- Diskvaccuum: http://hackaday.com/2014/01/10/a-diskvaccuum-for-obsolete-disk-formats

Sailfish { id: hakkerinMobiiliOS }

Viime joulukuussa julkaistiin ensimmäinen Sailfish-käyttöjärjestelmää ajava laite, Jolla. Nyt kun uutta älyluuria on ehditty jo hiplata jonkin aikaa, on aika perehtyä sen ohjelmointiin.

Teksti: Asser Lähdemäki Kuvat: Mitol Berschewsky, Asser Lähdemäki, Teemu Likonen
Esimerkkisovellus: Teemu Likonen, Asser Lähdemäki

Sailfish OS on tuore käyttöjärjestelmä mobiilimarkkinoilla, mutta sen juuret ovat Nokian Meego-järjestelmässä. Kun Nokiassa päätettiin siirtyä Windows-käyttöjärjestelmään, Meegon kehittäjät kokivat, että heidän työnsä valui hukkaan. Monet päättivät lähteä Nokiasta vapaaehtoisesti ja perustaa uuden yrityksen, Jollan.

Uuden yrityksen tavoitteena oli tuoda mobiilimarkkinoille jotakin ennennäkemätöntä ja rikkoa status quo. Sitä varten kehitettiin oma Sailfish-käyttöjärjestelmä, joka erottuu markkinoiden muiden pelurien järjestelmistä. Käyttöjärjestelmäänsä varten yhtiö kehitti myös Jolla-matkapuhelimen, mutta tarkoituksena on, että Sailfishin voi asentaa muidenkin valmistajien laitteisiin.

Kun nyt markkinoilla kerran on ihan uudenlainen järjestelmä, kannattaa siihen vähän perehtyä. Tässä artikkelissa emme painele uuden laitteen nappuloita vaan tutkimme sen käyttöjärjestelmää kehittäjän näkökulmasta. Millaisista osista se rakentuu? Miten sille tehdään omia ohjelmia? Omaa Jolla-puhelinta ei välttämättä tarvita, sillä Sailfishin kehitysympäristöön kuuluu virtuaalikone, jossa artikkelin esimerkkisovellusta voi kokeilla.

Purjekalan anatomia

Sailfish OS on suunniteltu toimimaan laitteissa, jotka tukevat Android-käyttöjärjestelmää. Niinpä pohjimmaisena Sailfishin ohjelmistopinossa ovat Linux-ydin ja Androidin ajurit. Ajureita käytetään libhybris-kirjaston avulla.

Seuraavat ohjelmistokerrokset ovat Mer, Sailfish UI ja Nemo. Mer perustuu Nokian Meego-järjestelmään. Se sisältää ohjelmistoja, jotka toimivat Linux-ytimen ja käyttöliittymän välissä. Mer tarjoaa käyttöliittymätason ohjelmille rajapinto-ja ytimen palveluihin. Ohjelmistoja ovat esimerkiksi Systemd, Dbus ja Wayland. Merin päälle rakentuu Sailfish UI, joka tarjoaa ohjelmien käyttöliittymän. Sailfish UI on toistaiseksi pääosin suljettua koodia, mutta se hyödyntää myös avoimen Nemo-projektin osia. Taulukkoon 1 on koottu Sailfish-järjestelmän eri kerrokset ja niiden tehtävät.

Tällaisen ohjelmistopinon varaan alamme rakentaa omaa esimerkkisovellustamme. Ohjelmointiin käytämme Qt-sovelluskehystä ja sen deklaratiivista käyttöliittymän kuvauskieltä, QML:ää. Mikäli kieli ei ole lukijalle tuttu, kannattaa tutustua Skrollin numeroissa 2013.1 ja 2013.3 julkaistuihin QML-artikkeleihin tai hankkia perustiedot muualta, esimerkiksi internetistä osoitteesta http://qmlbook.org/. QML perustuu Javascriptiin, joten sitäkin täytyy vähän osata.

Kehitysympäristö haltuun

Ensimmäinen askel Sailfish-ohjelmointiin on kehitystyökalujen (software develop-

Osa	Kerros	Tehtävä	Koodin saatavuus
Linux-ydin, Android-ajurit ja libhybris	ydin	käyttöjärjestelmän ytimen tehtävät, oheislaitteiden kanssa kommunikointi	Linux-ydin on avointa koodia, samoin libhybris. Ajurit ovat yleensä suljettuja.
Mer	väliohjelmistot, taustapalvelut	Tarjoaa sovelluksille mm. rajapinnan ytimen ja ajureiden palveluihin.	Mer-projekti on täysin avointa koodia.
Sailfish UI, Nemo	käyttöliittymä	Toteuttaa käyttöliittymän laitteelle ja käyttöliittymäkomponentit sovelluksille.	Nemo on avoin, Sailfish UI ei.
sovellukset	sovelluskerros	Tarjoaa käyttäjille hyödyllisiä toimintoja ja sisältöä.	tapauskohtaista

Taulukko 1. Sailfish OS:n ohjelmistopino.

ment kit, SDK) lataaminen ja asentaminen. Sailfish SDK on saatavilla Linux-, Windows- ja OS X -käyttöjärjestelmille osoitteessa https://sailfishos.org. Sivuilla on paljon hyvää luettavaa kehitysympäristön asennuksesta ja käytöstä. Lisäksi on asennettava virtualisointiohjelmisto Virtualbox, joka on saatavissa osoitteesta https://www.virtualbox.org.

Kehitysympäristö koostuu Qt Creator -työkalusta sekä MerSDK- ja Sailfish Emulator -virtuaalikoneista. Qt Creatorilla hallitaan sovellusprojektia ja käsitellään itse ohjelmakoodia. Virtuaalikoneet ovat ohjelmien kääntämistä ja testaamista varten.

Kääntämisessä virtuaalikoneen hyvä puoli on se, ettei käännöstyökaluista tarvitse olla erikseen versiota jokaiselle alustalle. Toisaalta kääntäminen on hitaampaa, sillä virtuaalikoneen suorituskyky on heikompi kuin isäntäkoneen. Ohjelmien testaamisessa virtuaalikoneen etuna on, ettei puhelinta tarvitse kaivaa esiin joka välissä vaan ohjelmaa voi kokeilla samassa tietokoneessa.

Qt Creatorissa luodaan uusi Sailfish-sovellusprojekti valitsemalla valikosta File → New File or Project. Avautuvasta ikkunasta valitaan ”Sailfish OS Qt Quick Application” ja annetaan projektille nimi ja sijainti levyllä. Seuraavassa vaiheessa on hyvä valita ohjelman kohdealustaksi sekä ”i486-x86” että ”armv7hl”. Ensin mainittu on Sailfish-emulaattorille ja jälkimmäinen oikeaa mobiililaitetta varten. Tämän jälkeen ohjelma pyytää vielä kuvausta ja muita tietoja, joita käytetään pohjana sovelluspaketin luomisessa. Jos sovelluksen haluaa julkaista Jollan sovelluskaupassa, on nimessä käytettävä muotoa ”harbour-minunsovellus”.

Qt Creator luo automaattisesti projektipohjan, jossa on yksinkertainen C++-ohjelma ensimmäisen QML-tiedoston lataamiseksi. Projektissa on valmiina muutamia QML- ja muita tiedostoja, jotka antavat esimerkin Sailfish-sovelluksen rakenteesta.

Kuva 1. Qt Creatorin työkalupalkki.

Kun sovellusprojekti on avattu, ilmestyy Qt Creatorin vasempaan työkalupalkkiin kuvan 1 mukaiset painikkeet. Kuvassa näkyvän a-painikkeen avulla valitaan esimerkiksi sovelluksen kohdealusta (i486 tai armv7hl). Painikkeita b, c ja d käytetään ohjelman kääntämiseen ja suorittamiseen, ja painikkeilla e ja f käynnistetään ja sammutetaan virtuaalikoneet.

Ensin kannattaa käynnistää virtuaalikoneet. MerSDK-virtuaalikone on ohjelmien kääntämistä varten, ja Sailfish OS -emulaattori on tarkoitettu niiden testaamiseen. Toisinaan virtuaalikoneet käynnistyvät sen verran hitaasti, että Qt Creator ilmoittaa aikakatkaisusta. Tällöin yleensä riittää, kun painaa uudestaan virtuaalikoneen käynnistyspainiketta.

Kun molemmat virtuaalikoneet on saatu käyntiin, kokeillaan Qt Creatorin luomaa projektipohjaa. Se on nimittäin toimiva Sailfish-ohjelma. Valitaan kohdealustaksi ”i486” eli Sailfish-emulaattori ja koontiversioksi ”Release”. Qt Creator antaa valittavaksi myös tavan, jolla käännetty sovellus siirretään kohdelaitteeseen. Vaihtoehto ”Deploy By Copying Binaries” kopioi käännetyt binääritiedostot ja projektin QML-tiedostot kohdejärjestelmän kansioon ”/opt/sdk/sovelluksen_nimi/”. Vaihtoehto ”Deploy As RPM Package” puolestaan tekee asennuspaketin, joka sitten siirretään ja asennetaan kohteeseen. Testausvaiheessa lienee järkevintä siirtää vain binääritiedostot. RPM-asennuspakettia tarvitaan lopullisessa versiossa, jotta sovellus rekisteröityisi oikealla tavalla Sailfish-järjestelmään.

Mikäli kaikki on sujunut niin kuin pitääkin, valitsemalla Run-toiminnon (ks. kuva 1, painike b) ohjelma ensin käännetään, siirretään sitten automaattisesti Sailfish-virtuaalikoneeseen ja käynnistetään siellä. Ohjelman kehitys ja testaus on siis varsin nopeaa, sillä kehitysympäristö huolehtii tiedostojen siirtämisestä ja asentamisesta kohdelaitteeseen.

Kehitysympäristö osaa siirtää ohjelmat myös Jolla-puhelimeen tai muuhun Sailfish-laitteeseen. Laitteen asetuksista on ensin kytkettävä kehitystila päälle valitsemalla Asetukset → Järjestelmä → Kehitystila. Hyväksy asetuksista myös etäyhteys ssh:n kautta ja aseta salasana.

Ulkoisen Sailfish-laitteen olemassaolosta on kerrottava myös kehitysympäristölle. Se tapahtuu valitsemalla Qt Creatorin valikosta Tools → Options… → Devices → Add. Valitse ”Mer ARM device” ja noudata ohjatun asetustoiminnon ohjeita. Asetuksiin täytyy muun muassa asettaa IP-osoite, josta Sailfish-laitteen tavoittaa. Kun laite on määritetty, saadaan omat sovellukset käännettyä ja siirrettyä sinnekin helposti. Käännöksen kohdealustaksi on tällöin valittava ”armv7hl”.

Tämän artikkelin kirjoitushetkellä kehitysympäristöstä ei ole vielä virallista julkaisua vaan ainoastaan alfaversio. Se voi tarkoittaa, että kaikki ei aina toimi ihan täydellisesti. Jos edellä mainittujen toimintojen kanssa on ongelmia, ratkaisua voi etsiä osoitteesta https://sailfishos.org/wiki/SDK_Alpha_Known_Issues.

Käyttöliittymä à la Sailfish Silica

Tietokonejärjestelmissä yhtenäinen, sujuva käyttökokemus syntyy siitä, että ohjelmat muistuttavat toisiaan ja niitä käytetään hyvin samalla tavalla. Sen vuoksi myös Sailfish-järjestelmä tarjoaa oman käyttöliittymäkirjastonsa, Sailfish Silican, jota hyödyntämällä ohjelmoija voi helposti luoda tyyliin sopivia sivuja, valikoita, painikkeita ja muita toimintoja. Ohjelmakirjaston lisäksi on hyvä perehtyä myös muihin käyttöliittymän suunnittelun periaatteisiin.

Sailfish-sovellusten käyttöliittymä perustuu eräänlaiseen sivuhierarkiaan. Sivut asetetaan pinoon (PageStack), ja sovelluksessa navigoidaan pinorakenteen avulla. Pinon päällimmäinen sivu on yleensä näkyvissä. Hierarkiassa ylemmän tason sivulle pääsee pyyhkäisemällä nykyinen sivu oikealle, jolloin sivu yleensä katoaa ja poistuu pinosta. Toisinaan on mahdollista myös lisätä uusi sivu näkyvän sivun päälle eli käyttöliittymässä oikealle puolelle. Tällöin pyyhkäisy vasemmalle näyttää uuden sivun.

Toinen erityispiirre Sailfishissä ovat sovellusten kannet. Taustalla toiminnassa olevat sovellukset näkyvät kotinäkymässä kantena, jota koskettamalla pääsee sovellukseen takaisin. Kannessa voi olla näkyvissä sovelluskohtaista tietoa. Esimerkiksi kalenterisovellus voi näyttää päivämäärän ja tapahtumia. Kannessa voi olla myös toimintoja, jotka aktivoidaan pyyhkäisemällä kantta oikealle tai vasemmalle. Niiden avulla ohjelman toimintaan voi vaikuttaa, vaikka se toimiikin taustalla piilossa.

Kolmas huomionarvoinen asia ovat Sailfishin tunnelmat. Ne sisältävät esimerkiksi taustakuvan, joka vaikuttaa käyttöliittymän väreihin. Soittoäänet ja muut hälytysasetukset voi määrittää tunnelmakohtaisesti. Tunnelmat ovat nopeasti vaihdettavissa, ja sovellusten täytyy ottaa se huomioon.

Sailfishin käyttöliittymäkomponentit ovat Sailfish Silica -kirjastossa, mutta sovelluksissa voi hyödyntää kaikkia QML:n peruskomponentteja. Useimmat Silican komponentit pohjautuvat QML:n peruskomponentteihin, joten Silican ohjekirjoissa ei ole aina mainittu peruskomponenteilta perittyjä ominaisuuksia. Ne

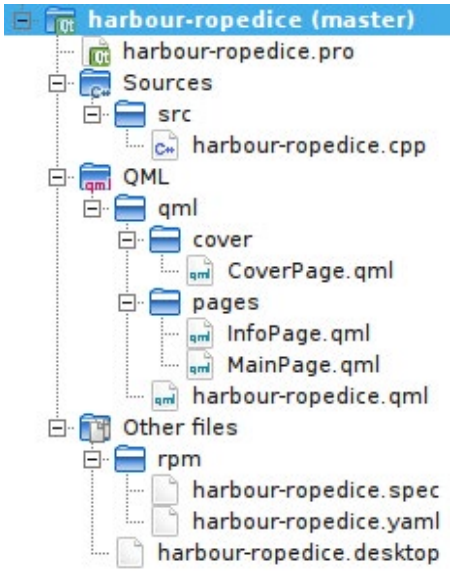
Essential types	Sailfish-sovelluksen perustoiminnot muun muassa sovellusikkunalle, sivupinolle, tunnelmien huomioimiseen ja leikepöydän käytölle.
Controls	Käyttöliittymän painikkeita, valitsimia, kytkimiä yms.
Text display and input	Tekstin esittämiseen ja syöttämiseen tarkoitetut komponentit.
Pages, views and containers	Sisältää komponentit sivujen esittämistä varten sekä eräitä muita käyttöliittymän rakenteita, kuten lista- ja ruudukkonäkymät.
Animations	Animaatiot erilaisille käyttöliittymän tapahtumille.
Menus	Alas- ja ylösvetovalikoiden komponentit sekä pikavalikkotoiminto.
Dialogs	Valitsimet väreille, ajalle ja päivämäärälle sekä yleinen kyselysivu.
Application covers	Sovelluskansien käsittelyyn tarvittavat komponentit.

Taulukko 2. Sailfish Silica 1.0 -kirjaston kategoriat.

löytyvät kuitenkin helposti Qt:n oppaista. Kannattaa varmistaa, että käytössä on opas vähintään Qt:n versiolle 5.1, sillä internetin hakukoneet antavat osumia myös versiolle 4.8.

Silica-kirjasto sisältää komponentteja ja ominaisuuksia sivupinon hallintaan, valikoiden tekemiseen, käyttöjärjestelmän tunnelmien huomioimiseen ja sovelluskannen käsittelyyn. Mukana on tietenkin myös erilaisia vipuja, painikkeita ja syötekomponentteja. Näitä tarvitsemme esimerkkisovelluksessamme. Lisäksi kirjasto tarjoaa rajapintoja Sailfish-järjestelmään, esimerkiksi leikepöydän käyttöön ja tiedostojärjestelmän standardikansioihin. Taulukkoon 2 on koottu Sailfish Silica 1.0 -kirjaston eri osa-alueet.

Tämän artikkelin esimerkkisovelluksen Silica-komponentit esitellään sitä mukaa, kun ne tulevat vastaan. Silican omat komponentit mainitaan erikseen, mutta muut ovat tavallisia QML:n komponentteja. Silicaan voi tutustua myös Qt Creatorin Welcome-välilehdellä olevasta kohdasta Examples, josta voi ladata esimerkkiprojektin nimeltä ”Sailfish Silica Component Gallery”. Projektissa esitellään komponenttien käyttöä ja myös niiden ulkoasu on nähtävissä.



Kuva 3. Sailfish-projektin kansiorakenne.

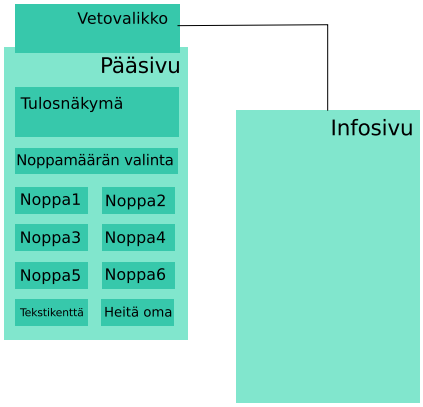
Rope dice – kun nopat jäivät kotiin

Lautapelien ystävän ja pöytäroolipelaajan vakiovarusteisiin kuuluu tietysti noppasetti. Mutta entä jos nopat pääsevät unohtumaan? Pelitapahtuma ei välttämättä mene pilalle, jos on osannut ohjelmoida puhelimeensa nopanheittosovelluksen. Älypuhelinhan on sentään kaikilla aina (?) mukana. Rakennamme yksinkertaisen mutta käytännöllisen nopanheittosovelluksen Sailfishille. Samalla sovellus esittelee Sailfish- ja QML-ohjelmoinnin perusasioita.

Sovellukseen tulevat omat painikkeensa nopille 4, 6, 8, 10, 12 ja 20, koska niitä tarvitaan usein. Lisäksi käyttäjä voi syöttää yhdelle nopalle sivumäärän ja heittää sitä, koska toisinaan tarvitaan myös esimerkiksi 100-sivuista noppaa. Joskus on hyödyllistä heittää useita noppia samanaikaisesti, joten myös noppien lukumäärän tulee olla valittavissa.

Sovellukseen tarvitaan pääsivu, joka avautuu ensimmäisenä ja jossa varsinaiset nopanheittotoiminnot ovat. Lisäksi luodaan infosivu, jossa kerrotaan tekijöiden nimet ja muuta tietoa sovelluksesta. Pääsivulta siirrytään infosivulle alasvetovalikon avulla. Käyttöliittymän rakenne on havainnollistettu kuvassa 2.

Siirrytään pikku hiljaa koodin pariin. Emme julkaise kaikkia esimerkkisovelluksen tiedostoja lehden sivuilla vaan ainoastaan tärkeimmät osat. Koko ohjelman voi ladata Skrollin sivuilta osoitteesta http://skrolli.fi/2014.2/. Avaa projektin .pro-tiedosto Qt Creatorissa ja valitse kohdealustoiksi ”i486” ja ”armv7hl”. Näin sovellus saadaan käännettyä sekä Sailfish-emulaattorille että oikealle laitteelle.



Kuva 2. Sovelluksen sivurakenne ja pääsivun sisältö.

Noppasovelluksen kansioiden rakenne näkyy kuvassa 3 ja osien merkitykset on selitetty taulukossa 3. Rakenne perustuu hyvin pitkälle Sailfish-kehitysympäristön luomaan projektipohjaan, joten se edustanee aika tyypillistä Sailfishin QML-sovellusta.

Lehden sivuilla olevissa ohjelmalistauksissa on eräissä kohdissa kolme pistettä (...). Merkintä tarkoittaa, että kohdasta on jätetty pois selitettävän toiminnallisuuden kannalta epäoleellista koodia. Nämä kohdat täydennetään kyllä artikkelin myöhemmissä listauksissa.

Ohjelman runko

Listauksessa 1 on sovelluksen pääosa, joka nivoo muut osat yhteen. Alkuriveillä on tuotu Qt Quickin peruskomponenttikirjasto ja Sailfish Silica -kirjasto. Lisäksi on tuotu qml-hakemiston alihakemistot pages ja cover. Hakemiston tuominen antaa käyttöön kaikki hakemiston qml-tiedostoissa määritellyt komponentit.

Sovelluksen käyttöliittymän rakennus alkaa siitä, että määritetään ApplicationWindow-komponentti eli sovellusikkuna.

harbour-ropedice.pro	Qt:n projektitiedosto, joka muistuttaa Makefilea.
src-kansio	Kansio C++-lähdekoodille.
harbour-ropedice.cpp	Automaattisesti luotu QML:n latauskoodi.
qml-kansio	Kansio QML-lähdekoodille.
CoverPage.qml	Sovelluskannen määrittelytiedosto.
InfoPage.qml	Infosivu, joka kertoo muun muassa sovelluksen tekijöistä.
MainPage.qml	Sovelluksen pääsivu, jossa nopanheittotoiminnot ovat.
harbour-ropedice.qml	QML-tiedosto, jossa asetetaan käytettävä aloitussivu ja sovelluskansi.
rpm-kansio	RPM-paketin luomiseen tarvittavat tiedostot.
harbour-ropedice.desktop	Tiedostossa muun muassa määritellään, miten sovellus käynnistetään ja mitä kuvaketta sovellus käyttää.

Taulukko 3. Sailfish-projektin kansiot, tiedostot ja niiden merkitys.

```
import QtQuick 2.0
import Sailfish.Silica 1.0
import "pages"
import "cover"

ApplicationWindow {
  id: base

  property string program_name: "Rope dice"
  property int button_width: width * (3 / 7)

  MainPage {
    id: mainPage
  }

  initialPage: mainPage

  cover: Component {
    CoverPage { ... }
  }
}
```

Listaus 1. Tiedosto harbour-ropedice.qml, josta käyttöliittymän määrittely alkaa.

Sen tärkeimpiin funktioihin kuuluvat activate ja deactivate, joiden avulla sovelluksen voi siirtää taustalle ja palauttaa aktiiviseksi. Komponentin kolme tärkeintä ominaisuutta ovat initialPage, cover ja pageStack.

Sivupinosta huolehtiva pageStack-ominaisuus alustetaan sisäisesti, joten se ei vaadi ohjelmoijalta toimenpiteitä. Ominaisuus initialPage asettaa aloitussivun, jonka täytyy olla Page tai vastaava komponentti. Sovellusikkunan cover-ominaisuus puolestaan kertoo, mitä komponenttia käytetään sovelluskannessa.

Nopanheittosivu

Listauksessa 2 kuvataan sovelluksen pääsivun rakenne eli sen sisältämät komponentit ja niiden hierarkia. Tiedoston juurikomponentti on Silica-kirjaston Page. Sen avulla voidaan muun muassa määrittää, tuetaanko näytön pysty- vai vaaka-asentoa vai molempia. Hyödyllinen ominaisuus on myös status, jonka antaa tietoa sivun tilasta, esimerkiksi kun käyttäjä siirtyy sivulle tai poistuu sivulta.

Seuraavaksi listauksessa määritellään SilicaFlickable-komponentti, joka täyttää koko pääsivun. Komponentti on QML:n Flickable-komponentin Sailfish-tyylitelty versio. Sen tärkein tehtävä on komponentin sisällön liikuttelu. Komponentille pitää muistaa asettaa contentHeight- ja contentWidth-arvot, jotta liikuttelu toimii oikein.

Liikuteltavien sisältöjen kanssa tarvitaan joskus vierityksen ilmaisinta eli jotakin Silican ScrollDecorator-komponenttia. Se onnistuu helpoimmin, kun asettaa SilicaFlickable-tyyppisen komponentin alikomponentiksi ScrollDecorator-komponentin, kuten on tehty listauksessa 2. Tällöin veritystä ilmaiseva komponentti automaattisesti tietää, minkä komponentin liikuttelua sen tulisi havainnollistaa.

PullDownMenu on Silican alasvetovalikko, ja sen on aina oltava SilicaFlickablen, SilicaListView'n tai vastaavan komponentin alikomponentti. Valikon sisältö ilmaistaan MenuItem-komponenttien avulla. Listauksessa 2 on määritetty valikkoon yksi kohta "Info", joka lisää sivupinon päällimmäiseksi InfoPage-komponentin. Ohjelmoijan on pidettävä huolta, että sivupinoon lisätään vain Page-komponentteja tai sen johdannaisia kuten Dialog-komponentteja. Vaihtoehtona alasvetovalikolle voi käyttää ylösvetovalikkoa eli PushUpMenu-komponenttia.

Sailfishin valikoiden suunnittelussa kannattaa huomioida, että niihin pääsee käsiksi vain silloin, kun ollaan sivun sisällön ylä- tai alareunassa. Valikoihin ei kannata laittaa kovin monta toimintoa, sillä muuten niitä on epämukavaa käyttää.

Listauksessa 2 seuraavana on Column-komponentti, joka on yksi QML:n sijoittelukomponenteista. Se kasaa alikomponenttinsa allekkain. Vastaavasti Row-komponentin avulla voidaan sijoittaa alikomponentit vierekkäin ja Grid-komponentilla ruudukkoon. Nämä ovat QML:n peruskomponentteja ja oikein käteviä myös Sailfish-sovelluksissa.

Sijoittelukomponentit rajoittavat alikomponenttiensa kohdistusankkurien käyttöä. Tämä on tietysti luonnollista, koska sijoittelukomponentin on tarkoitus vastata alikomponenttiensa sijoittelusta. Esimerkiksi listauksessa 2 olevan Columnin saa hajoamaan, kun määrittää jollekin alikomponentille ominaisuuden "anchors.top: parent.top". Ohjelma antaa siitä myös virheilmoituksen Qt Creatorin tulostusikkunaan.

Column-komponentille on määritetty Theme-olion avulla alikomponenttien välit ja marginaalit. Theme-olioita kannattaa hyödyntää, kun käyttöliittymässä määritellään esimerkiksi fontteja tai elementtien välejä. Niiden avulla ohjelman saa toimimaan sujuvasti Sailfish-laitteen erilaisilla tunnelma-asetuksilla.

Columnin ensimmäinen alikomponentti on PageHeader, joka luo sivulle Sailfish-tyylisen otsikon. Muiden alikomponenttien sisältöä on jätetty listauksesta pois, mutta niitä alamme tutkailla seuraavaksi.

Tekstiä, vipuja ja nappuloita

Siirrytään tutkimaan noppasovelluksen pääsivun tärkeimpiä komponentteja eli nopanheittotuloksesta kertovia tekstikomponentteja sekä nopanheittopainikkeita. Listauksessa 3 on pääsivun kaksi Label-komponenttia ja Slider-komponentti. Label on Silican oma tekstielementti, joka periytyy QML:n peruskomponentista Text. Label asettaa automaattisesti tekstin värin ja fontin.

Slider puolestaan on liukuvalitsin. Noppasovelluksessa sen avulla valitaan heitettävien noppien lukumäärä. Sliderille on asetettu koodissa arvoalue yhdestä

```
import QtQuick 2.0
import Sailfish.Silica 1.0

Page {
  id: root

  property string lastThrow: ""
  property bool diceIsThrown: false
  property int lastDiceMax: 0

  function roll_dice (max) { ... }

  SilicaFlickable {
    anchors.fill: parent

    PullDownMenu {
      MenuItem {
        text: "Info"
        onClicked: pageStack.push(Qt.resolvedUrl("InfoPage.qml"))
      }
    }

    VerticalScrollDecorator {}

    contentHeight: column.height

    Column {
      id: column
      spacing: Theme.paddingLarge
      anchors.margins: Theme.paddingLarge
      anchors.horizontalCenter: parent.horizontalCenter

      PageHeader {
        title: base.program_name
      }

      Label { ... }

      Label { ... }

      Slider { ... }

      Grid {
        columns: 2
        rows: 4
        spacing: Theme.paddingLarge
        rowSpacing: Theme.paddingLarge

        Repeater { ... }
        TextField { ... }
        Button { ... }
      }
    }
  }
}
```

Listaus 2. Sovelluksen pääsivun eli MainPage.qml:n rakenne.

```
Label {
  id: result
  text: " "
  font.pixelSize: Theme.fontSizeHuge
  horizontalAlignment: Text.AlignHCenter
  width: parent.width
}

Label {
  id: dice_rolls
  text: " "
  font.pixelSize: Theme.fontSizeExtraSmall
  horizontalAlignment: Text.AlignHCenter
  width: parent.width
  height: 20
  wrapMode: Text.Wrap
  maximumLineCount: 2
}

Slider {
  id: nod
  label: "Number of dice"
  minimumValue: 1
  maximumValue: 10
  stepSize: 1
  value: 1
  valueText: value
  width: parent.width
}
```

Listaus 3. Sovelluksen pääsivun Label- ja Slider-komponenttien sisältö.

kymmeneen. Nykyinen arvo kerrotaan sovelluksen käyttäjälle valueText-ominaisuuden avulla. Sen arvo viittaa suoraan value-ominaisuuteen, jonka arvo puolestaan päivittyy automaattisesti, kun liukua siirretään.

Loput pääsivun toiminnot ovat listauksessa 4. Grid-komponentin avulla napit sijoitetaan ruudukkoon, jossa on kaksi saraketta ja neljä riviä. Grid sijoittaa alikomponenttinsa järjestyksessä vasemman ylänurkan solusta alkaen. Koska ruudukossa on yhteensä kahdeksan solua, pitää Grid-komponentilla olla kahdeksan alikomponenttia. Listauksesta kuitenkin nähdään, että niitä on vain kolme. Ruudukko täytetäänkin Repeater-komponentin avulla.

Repeater luo jokaista mallin alkiota kohden yhden edustajan (delegate). Mallin dataan päästään käsiksi modelData-ominaisuuden avulla, jota voi käyttää edustajan määrittelyssä. Lisäksi edustajassa voi käyttää ominaisuutta index, joka kertoo data-alkion sijoituksen mallissa. Repeater-komponentti käytännössä korvaa sen, että painikekomponentti Button olisi toistettu kuusi kertaa hieman erilaisella nopan sivumäärällä.

Button-komponentti on Silican painike. Yleensä sille asetetaan kolme ominaisuutta: leveys, teksti ja suoritettava toiminto, kun painallus havaitaan. Tässä tapauksessa painalluksen seurauksena kutsutaan funktiota roll_dice, joka arpoo noppien silmäluvut ja sijoittaa arvot Label-komponentteihin. Funktio näkyy listauksessa 5.

Listauksessa 4 olevat kaksi viimeistä komponenttia muodostavat käyttäjän määrittämän nopan. Button-komponentti on muuten samanlainen kuin edelläkin, mutta nopan sivumäärä haetaan tällä kertaa erillisestä TextField-komponentista ja sen text-ominaisuudesta.

TextField perii QML:n peruskomponentin TextInput, joten niiden ominaisuudet vastaavat pääosin toisiaan. Esimerkiksi syötteentarkistus voidaan hoitaa samalla tavalla eli käyttämällä IntValidator-, DoubleValidator- tai RegExpValidator-komponenttia. Noppasovelluksessa halutaan kokonaislukuja, joten käytetään IntValidatoria ja määritellään lukualueeksi 2–10 000.

TextField-komponentin hyödyllisiä ominaisuuksia ovat myös inputMethodHints ja placeholderText. Näistä ensimmäinen antaa virtuaalinäppäimistölle ohjeita, millaista syötettä odotetaan. Esimerkkisovelluksessa käytetty Qt.ImhDigitsOnly kertoo, että tekstikenttä hyväksyy vain numeroita, joten Sailfish tuo esiin numeronäppäimistön, kun tekstikenttä aktivoidaan. Ominaisuus placeholderText puolestaan asettaa tekstin, joka näytetään tekstikentän ollessa tyhjänä. Sen avulla voi vihjata käyttäjälle, mitä sisältöä kenttään halutaan.

Sovelluskansi – vuorovaikutteinen kiiltokuva

Katsotaan seuraavaksi, miten sovelluskansi toteutetaan. Kansi otetaan käyttöön ApplicationWindow-komponentin cover-ominaisuuden avulla. Suositeltavaa on käyttää kannen toteutuksessa CoverBackground-komponenttia, joka sisältää esimerkiksi logiikkaa kannen sijoittamiseen kotinäkymässä.

Noppasovelluksen kanteen haluamme viimeisimmän nopanheiton tuloksen sekä pyyhkäisytoiminnon, jolla noppaa voi heittää suoraan kannesta käsin. Listauksessa 6 on varsinainen kannen määrittely. Listauksen alussa määritellään pari ominaisuutta ja signaali, jotka auttavat pyyhkäisytoiminnon toteutuksessa. Kannessa näytettävät tekstit on toteutettu Label-komponenteilla, jotka puolestaan ovat sijoittelusta vastaavan Column-komponentin sisällä.

Kannen pyyhkäisytoiminnot määritellään CoverActionList-komponentin alikomponentteina, joita voi olla enintään kaksi. Ensimmäinen CoverAction-alikomponentti tulee näkyviin kannen

```
Grid {
  columns: 2
  rows: 4
  spacing: Theme.paddingLarge
  rowSpacing: Theme.paddingLarge

  Repeater {
    model: [4, 6, 8, 10, 12, 20]
    delegate: Button {
      text: "D" + modelData
      onClicked: roll_dice(modelData)
      width: base.button_width
    }
  }

  TextField {
    id: custom_dice
    text: "100"
    placeholderText: "Sides count"
    validator: IntValidator { bottom: 2; top: 10000 }
    inputMethodHints: Qt.ImhDigitsOnly
    width: base.button_width
  }

  Button {
    text: "D" + custom_dice.text
    onClicked: roll_dice(custom_dice.text)
    width: base.button_width
  }
}
```

Listaus 4. Sovelluksen pääsivun loppuosa eli Grid-komponentti ja sen alikomponentit.

```
function roll_dice (max) {

  root.lastDiceMax = max

  var sum = 0
  var all_rolls = " "
  for (var i = 1; i <= nod.sliderValue; i = i + 1) {
    var roll = Math.floor((Math.random() * max) + 1)
    sum = sum + roll
    if (nod.sliderValue > 1) {
      if (i == 1) {
        all_rolls = roll.toString()
      } else {
        all_rolls = all_rolls + " + " + roll
      }
    }
  }

  result.text = sum
  root.lastThrow = nod.sliderValue + "d" + max + ": " + sum
  dice_rolls.text = all_rolls

  if(root.diceIsThrown === false) {
    root.diceIsThrown = true
  }
  return 0
}
```

Listaus 5. Nopan silmäluvun arvonnasta vastaava roll_dice-funktio.

```
CoverBackground {
  id: root

  property string lastThrow: ""
  property bool coverThrowEnabled: false

  signal requestThrow()

  Column {
    anchors.centerIn: parent
    Label {
      anchors.horizontalCenter: parent.horizontalCenter
      text: "Last throw:"
    }
    Label {
      anchors.horizontalCenter: parent.horizontalCenter
      text: root.lastThrow
    }
  }

  CoverActionList {
    enabled: root.coverThrowEnabled

    CoverAction {
      iconSource: "/usr/share/icons/hicolor/86x86/apps/harbour-ropedice.png"
      onTriggered: root.requestThrow()
    }
  }
}
```

Listaus 6. CoverPage.qml, sovelluskannen määritys.

```
cover: Component {
  CoverPage {

    lastThrow: mainPage.lastThrow
    coverThrowEnabled: mainPage.diceIsThrown

    onRequestThrow: {
      mainPage.roll_dice(mainPage.lastDiceMax);
    }
  }
}
```

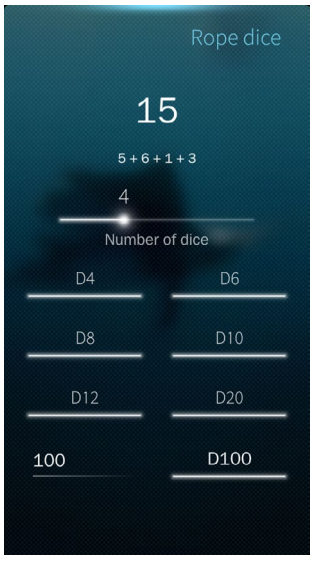
Listaus 7. Tiedosto harbour-ropedice.qml. Asetetaan sovelluskansi ja yhdistetään se pääsivun toimintoihin.

Lisää tuulta purjeisiin

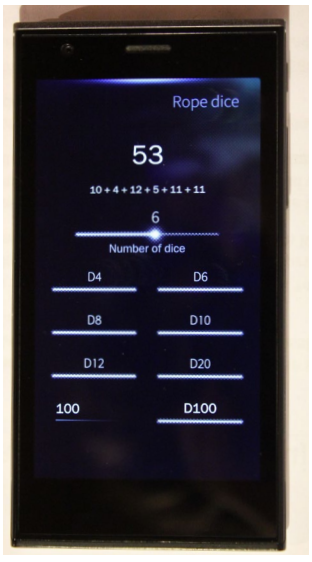
- Sailfish Silica -komponenttikirjasto: https://sailfishos.org/sailfish-silica/index.html.
- Sailfish-projekteja Githubissa: https://github.com/sailfishapps.
- Muita avoimen lähdekoodin projekteja: https://wiki.merproject.org/wiki/Sailfish_OS_OpenSource_Apps.
- Käyttäjien ylläpitämä, hitaasti päivittyvä kooste Sailfish-kehityksestä: https://github.com/hardcodes/developwithsailfishos.
- Usein kysyttyjä kysymyksiä: https://sailfishos.org/develop-faq.html.
- Esimerkkiohjelma ”Hello World Pro”, joka täyttää Jollan sovelluskaupan vaatimukset: https://github.com/amarchen/helloworld-pro-sailfish.
- Tietoa sovellusten asennuspakettien tekemiseen: https://sailfishos.org/develop-packaging-apps.html.
- Ohjeita kääntämiseen ilman Qt Creatoria: https://wiki.merproject.org/wiki/Platform_SDK_and_SB2.
- Sähköpostilista: https://lists.sailfishos.org/cgi-bin/mailman/listinfo/devel.
- Tietoa Jollan sovelluskaupasta: https://harbour.jolla.com/faq.
- Avoin julkaisualusta ilman laatuvaatimuksia: https://openrepos.net/.
- Irc-kanavalta #sailfishos (Freenode) saa vertaistukea. Jotta keskustelijat eivät kyllästyisi samoihin kysymyksiin, kannattaa vastauksia etsiä ensin lokeista esimerkiksi Googlella: ”<hakusanat> site: http://www.merproject.org/logs/”.

vasempaan alanurkkaan ja toinen oikealle. Kannen pyyhkäisy oikealle synnyttää ensimmäisen komponentin triggered-signaalin. Toinen CoverAction puolestaan reagoi, kun pyyhkäistään vasemmalle. Jos alikomponentteja on vain yksi, se sijoitetaan keskelle kannen alareunaa, ja triggered-signaali syntyy pyyhkäisyllä kumpaan suuntaan tahansa.

Aiemmin esittelimme listauksen 1, josta sovelluskannen määrittely oli jätetty pois. Puuttuva koodi on nähtävissä listauksessa 7. Siinä sidotaan kannen ominaisuudet päivittymään pääsivun ominaisuuksien mukaan ja määritellään käsittelijä requestThrow-signaalille. Käsittelijä asetetaan kutsumaan pääsivun roll_dice-funktiota, ja näin nopanheiton tulos päivittyy kanteen automaattisesti ominaisuuksien sitomisen ansiosta.



Valmis noppasovellus virtuaalikoneessa...



...ja ikiomassa puhelimessa.

Projektin seuraavat vaiheet

Artikkelissa käytiin lyhyesti läpi Sailfish-sovelluksen kehittämiseen tarvittavat työkalut ja QML-komponentit. Esimerkkisovellus on nyt valmis ja näkyy oheisissa kuvissa. Sovelluksen voisi periaatteessa julkaista Jollan sovelluskaupassa, sillä siinä on noudatettu kaupan teknisiä sääntöjä. Openrepos-julkaisualustalle sen voi lähettää säännöistä välittämättä.

Vaikka sovellus olisi julkaisukunnossa ja kehittäjän mielestä täydellinen, on käyttäjillä kuitenkin ikävä taipumus löytää puutteita. Nopanheittosovellukseen voisi lisätä vielä paljon ominaisuuksia. Alla on lueteltu jatkokehitysideoita, joiden avulla myös Sailfish-ohjelmointiin pääsee uppoutumaan syvemmin. Sulkeissa on vinkiksi komponentteja, joista voi olla hyötyä.

- Nopanheiton tuloksen kopioiminen leikepöydälle, esimerkiksi vetovalikon tai sovelluskannen pyyhkäisytoiminnon avulla (Clipboard, PullDownMenu, MenuItem, CoverActionList).
- Erillinen heittohistoriasivu, josta näkee edelliset heitot (ListModel, ListElement, SilicaListView, ListItem).
- Heittohistorian tyhjennystoiminto varustettuna katumusajastimella (RemorsePopup).
- Itse määritettävän nopan heittäminen virtuaalinäppäimistön rivinvaihtonäppäimellä (EnterKey).

Vaikka artikkelissa käsiteltiin vain QML:ää, kannattaa muistaa, että C++-kieli näyttelee vielä merkittävää osaa Sailfish-sovellusten kehityksessä. Monimutkaisiin sovelluksiin QML ei ehkä ole riittävä, ja siitä puuttuu joitakin Sailfish-järjestelmän rajapintoja. Pelkällä QML:llä päästiin kuitenkin hyvään alkuun, ja se kyllä riittää paljon pidemmällekin. Tämän pohjalta on hyvä jatkaa kohti monimutkaisempia sovelluksia.

Humanistin kolvauskoulu

Elektroniikka on mukava harrastus, ja sen perusteiden osaamisesta on paljon hyötyä. Humanistin kolvauskoulu on tarkoitettu sinulle, joka olet aina miettinyt, osaisitko rakentaa jotain.

Teksti: Mikko Heinonen
Kuvat: Risto Mäki-Petäys, Mikko Heinonen

Omat elektroniikan opintoni rajoittuvat kahteen viikkotuntiin peruskoulun 8. ja 9. luokalla yli 20 vuotta sitten. Uravalinta vei eri suuntaan, mutta olen silti lähes jatkuvasti huomannut tarvitsevani juottamisen kaltaista perustaitoa. On hyödyllistä osata tehdä vaikkapa erilaisia johtoja, ja usein säästää myös rahaa korjaamalla rikkoutuneita laitteita itse.

Juottamisessa ei sinällään ole kyse mystiikasta. Juottimen eli kolvin avulla lämmitetään juotostinaa niin, että se sulaa halutun liitoksen päälle. Kun juotin nostetaan pois, tina jäähtyy ja muodostaa sähköä johtavan liitoksen kahden kontaktipinnan välille. Juotos on liitoksena huomattavasti varmatoimisempi kuin pelkkään metallikontaktiin perustuvat kytkentätavat, eli erilaiset puristettavat liittimet tai pelkkä johtojen kietominen yhteen.

Sähkölaitteiden kanssa on tärkeää, ettei koske mihinkään sellaiseen, jota ei ymmärrä. Kouluttamattoman harrastajan on syytä pysyä erossa kaikenlaisesta, mikä liittyy pistorasiasta saatavaan verkkovirtaan. Siksi aloitamme helposta projektista ja teemme perinteisen digitaalisen peliohjaimen. Se on etenkin juotosten laadun suhteen huomattavasti suurpiirteisempää puuhaa kuin piirilevylle juottaminen.

Ilotikun anatomiaa

Niin sanottua Atari-liitäntäistä peliohjainta käytti suurin osa kotitietokoneista. Nimensä se sai Atari 2600 -pelikonsolin mukaan. Kuten 2600 itse, myös sen peliohjain oli hyvin yksinkertainen laite, joka koostui viidestä kytkimestä: ylös, alas, oikealle, vasemmalle ja tulitus. Jokaista suuntaa vastaa yksi peliohjainliitännän nasta, minkä lisäksi niillä on vielä yhteinen maanasta. Kytkemällä maa yhteen tietyn suunnan tai tulituksen kanssa saadaan aikaan haluttu toiminto.

Peliohjaimia on edelleen mahdollista ostaa kaupastakin, mutta niissä on yleensä vain Atarin perustoiminnot. Koska 9-napaisen liittimen nastoja jäi yli, muutamat valmistajat käyttivät niitä omiin tarkoituksiinsa. Esimerkiksi MSX:ssä on mahdollista käyttää kahta tulitusnäppäintä, Amigassa vieläkin enemmän.

Tässä esimerkissä rakennan MSX-joystickin, jossa siis kytketään kaksi tulitusnäppäintä. Muuten kytkentä on samanlainen kaikissa tietokoneissa. Tärkeää on kuitenkin huomata, ettei tällaista ohjainta kannata liittää Amigaan. MSX:n toisen tulitusnäppäimen linja on Amigassa kytketty 5 voltin jännitteeseen, jolloin sitä painettaessa syntyy oikosulku ja Amiga voi rikkoutua.

Saat ohjeet oman laitteesi peliohjaimelle helposti internetin hakukoneella. Hakemalla esimerkiksi ”msx joystick pinout” saa eteensä selkeän kuvan tarvittavasta nastajärjestyksestä. Liitin on numeroitu siten, että kun katsot sitä edestäpäin, nasta 1 on ylävasemmalla, nasta 9 alaoikealla.

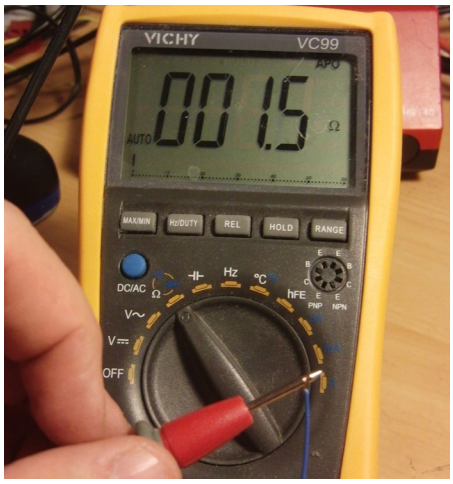
Välineet kuntoon

Ilman kunnollisia työkaluja homma loppuu alkuunsa. Tarvitset ainakin elektroniikalle tarkoitetun juottimen (mieluiten säädettävän juotinaseman), juotostinaa ja sivuleikkurit tai kuorimapihdit. Juottimen puhdistukseen tarvitaan joko vedellä kostutettu sieni tai metallisäikeistä tehty tyyny. Myös yleismittari kannattaa hankkia, sillä siitä on runsaasti iloa, eikä perusmalli paljon maksa. Pieni pala hiomapaperia helpottaa juotettavien pintojen valmistelua.

Juottimen lämpötilan pitää olla sopiva. Liian kuuma juotin sulattaa myös johtojen eristeet; liian viileä taas ei sulata edes tinaa. Hyvä peruslämpötila on noin 350 astetta. Paksu johto ja tina vaativat

Kuva 1. Tarvittavat työkalut.

Kuva 2. Tarveaineet.



Kuva 3. Yleismittari näyttää, että johdon päät ovat yhteydessä.

enemmän lämpöä kuin ohuet.

Kuvassa 2 näkyvät ohjaimen rakennusaineet. Periaatteessa sen voisi tehdä käyttämällä pelkkiä teollisuuskytkimiä, mutta käyttömukavuuden kannalta suosittelen hankkimaan peliautomaatteihin tarkoitetun ohjaimen rungon ja yksi tai kaksi nappia tarpeidesi mukaan.

Johdon teemme Atari-ohjaimille tarkoitetusta jatkojohdosta. Tähän on kaksi syytä. Ensinnäkin 9-napaisen liittimen juottaminen voi olla turhauttava projekti aloittelijalle. Toisekseen kaupoista saa yleensä vain ruuveilla varustettuja liittimiä, jotka eivät aina mahdu kotitietokoneiden ahtaisiin joystick-portteihin. Lisäksi tarvitset pätkän tavallista, 1-johtimista sähköjohtoa maanastan kytkemistä varten. Yleensä kotona on jokin rikkinäinen laite, josta piuhanpätkän voi hyödyntää.

Osat voi hankkia vaikkapa nettikaupoista. Perustasoinen ohjain maksaa noin 10 euroa, napit pari kolme euroa kappale ja kaapelin hinnaksi muodostuu reilu vitonen. Lisäksi tulee vielä kotelon hinta, ellet satu jo omistamaan sopivaa muovilaatikkoa. Ohjaimen voi hyvin koota esimerkiksi rikkoutuneen ulkoisen kiintolevyn koteloon ja nakata itse kiekon SER-keräykseen.

Nastat järjestykseen

Koska oikaisemme johdon valmistuksessa, meidän pitää ensin selvittää, miten jatkojohdon nastat on kytketty. Katkaise jatkojohdon urosliitin (se, jossa on metallisia nastoja, ei sitä, jossa on reikiä) sivuleikkureilla tai kuorimapihdeillä. Voit hävittää urosliittimen, sillä käytämme jatkossa vain naaraspäätä ja itse johtoa. Kuori ulompi eriste pois noin 20 cm:n matkalta. Kuori sitten sisempää eristettä kaikkien johtojen päästä 2–3 cm:n matkalta. Huolehdi, etteivät johdot kosketa toisiaan.

Nasta	MSX
1 ruskea	YLÖS
2 oranssi	ALAS
3 harmaa	VASEN
4 musta	OIKEA
5 punainen	-
6 keltainen	NAPPI 1
7 sininen	NAPPI 2
8 valkoinen	-
9 vihreä	MAA

Taulukko 1. Erään joystick-johdon nastajärjestys.

Ota yleismittari esiin ja kytke se vastusmittaukselle (merkkinä on oomega). Kääri pätkä juotostinaa mittarin toisen anturin ympärille. Työnnä sitten juotostinan pätkä jatkojohdon naarasliittimen reikiin yksi kerrallaan, alkaen ylävasemmalta (1) ja päätyen alaoikealle (9). Jokaisen reiän kohdalla liitä toiseen anturiin johtoja yksi kerrallaan, kunnes löydät sen, jonka kohdalla yleismittari herää eloon (eli näyttää vastusta, virta siis kulkee johdon läpi). Kirjaa ylös, minkä värinen johto vastaa mitäkin nastaa. Selvitä sen jälkeen haluamasi ohjaimen nastajärjestyksestä, mihin johdot pitää kytkeä. Taulukossa 1 on esimerkki Ebaysta ostetusta johdosta ja MSX-joystickista.

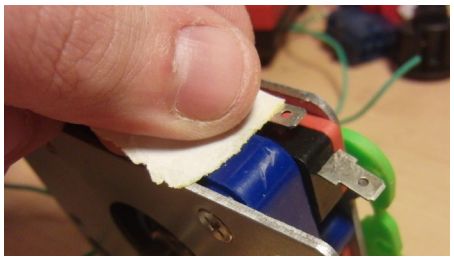
Ennen kuin aloitat juottamista, tee jatkojohtoon solmu suunnilleen 5 cm:n päähän siitä, mistä aloit kuoria ulompaa eristettä. Solmu toimii myöhemmin vedonpoistajana. Näet sen oikean paikan myöhemmistä kuvista.

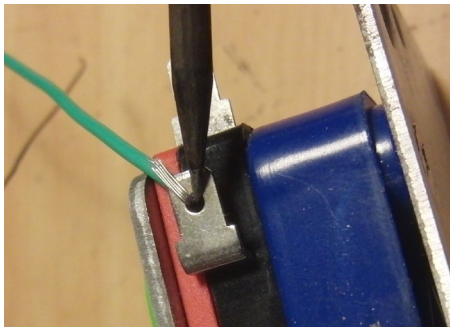
Piuhoitus alkaa

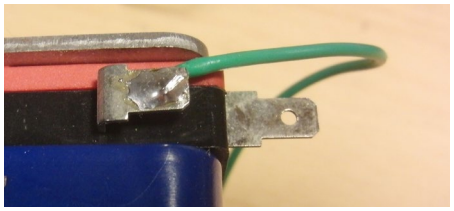
Laita juotin lämpiämään ja ota ohjainosa käteesi. Koska maataso tulee kiinni kaikkiin kytkimiin, on mielekästä juottaa kytkimet ensin kiinni toisiinsa ja sitten vasta varsinaiseen liittimeen. Käytän esimerkissä vihreää maajohtoa. Voit valita minkä värin tahansa, kunhan muistat, mikä se on.

Katso, millaisia kytkimiä hankkimasi ohjain käyttää. Jos napoja on vain kaksi, ei ole väliä, miten päin ne kytket. Jos napoja on kolme, ne on merkitty NO (normally open) ja NC (normally closed). Käytämme tässä kytkennässä NO-napoja. NC-navat ovat lepoasennossa kytkettynä, mutta me haluamme juuri päinvastaisen toiminnon.

Kytkimen kontaktipinta on todennäköisesti hapettunut ja siinä on myös varastorasvaa. Juotostina tarttuu erittäin huonosti tällaiseen pintaan. Ota pieni pala hiomapaperia ja karhenna pinta ennen kuin edes yrität juottaa. Vedä sitten johto pinnassa olevan reiän läpi ja asette-


Kuva 4. Karhenna pinnat ennen juottamista.


Kuva 5. Juotin paikallaan.


Kuva 6. Voitto!

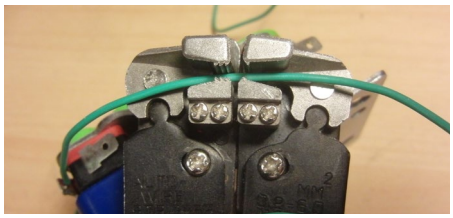
le se niin, että se pysyy paikallaan pitämättä – tarvitset seuraavassa molempia käsiäsi.

Paina juotin kiinni kontaktipintaan ja syötä toisella kädellä rullasta juotostinaa juottimen päälle niin, että se sulaa. Kun tina tarttuu johtoon ja karhennettuun pintaan, nosta juotin irti. Älä pidä juotinta kiinni pinnassa liian pitkään, ettei se johda lämpöä kytkimen sisään.

Jos noudatit ohjetta, olet nyt tehnyt ensimmäisen juotoksesi. Se ei ehkä ole järin kaunis, mutta jos tinaa on sekä johdossa että kontaktipinnassa eikä johto irtoa vetämällä, liitos on riittävän hyvä tähän tarkoitukseen. Tinaa ei kannata lisätä tarpeettoman paljon, sillä kontakti ei siitä ainakaan parane.

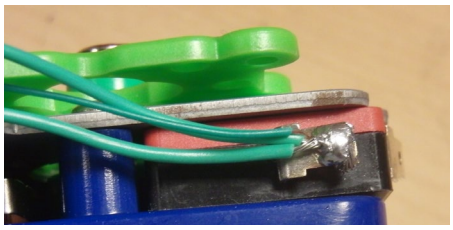
Kierrä nyt koko ohjain ja juota maajohto kiinni jokaisen kytkimen toiseen napaan. Jos käytössäsi on kunnolliset kuorimapihdit, voit oikaista hieman. Kuori maajohtoa keskeltä, vedä se läpi rei'istä kuorineen ja juota se sitten kiinni kuoritusta osasta. Näin johtoa ei tarvitse katkaista eikä kaikkiin napoihin tarvitse liittää kahta johtoa.

Neljänteen kytkimeen on juotettava kiinni myös kaksi muuta johtoa: liittimeltä tuleva maajohto ja tulitusnappien tarvitsema maasignaali. Kuori kaikkia johtoja noin 5 cm:n matkalta, kiedo ne yhteen, työnnä läpi kytkimen reiästä ja juota kiinni.

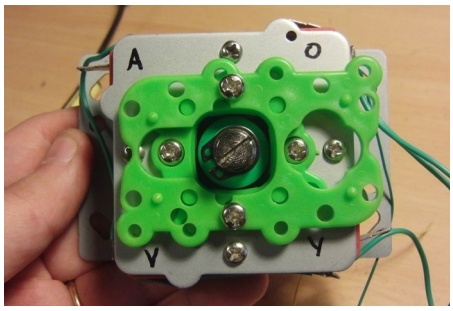


Kuva 7. Kuorimapihdit.

Kuva 8. Johto kuorittuna keskeltä.



Kuva 9. Kolmen johdon liitos.



Kuva 10. Kytkinten ohjaussuunnat alhaalta päin.

Suunnat selville

Seuraavaksi pitää selvittää, mihin kytkimiin signaalijohtimet liitetään. Käännä ensin ohjain siten, että sauva näyttää ylöspäin. Päätä sitten, mikä suunta tulee olemaan ylöspäin, ja merkitse se ohjaimeen.

Käännä nyt ohjain ylösalaisin ja tarkista, mitä kytkintä sen varsi painaa, kun painat tikkua ylöspäin. Tee sama muille suunnille ja merkitse suunnat ohjaimen pohjaan. Tämä on hyvin oleellista, sillä haluat varmasti pelihahmon tottelevan ohjausta oikein.

Juota nyt eri ilmansuuntien johtimet kiinni kytkimien napoihin. Periaate on sama kuin maajohdon kanssa: kuori, karhenna pinta, pujota johto, juota.

Koteloitumisvaihe

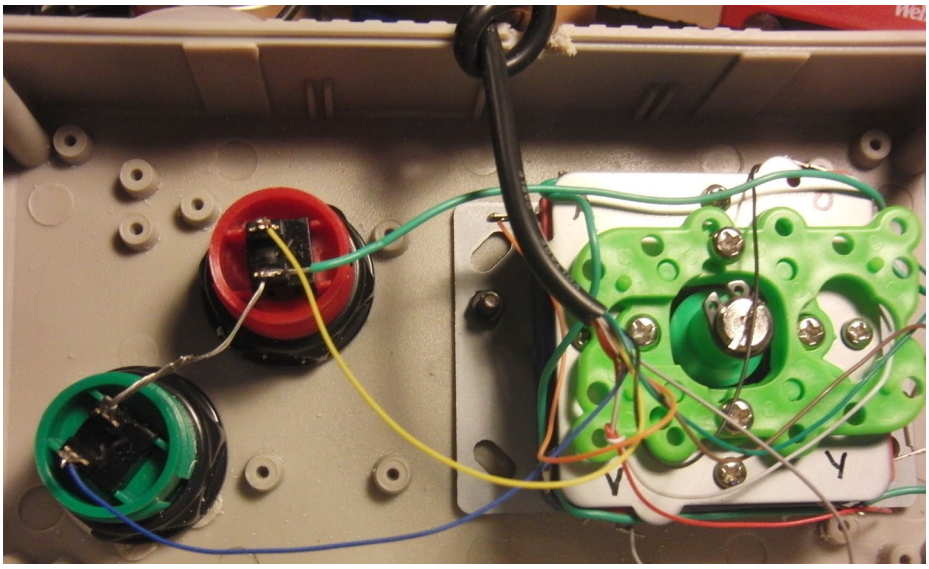
Ennen nappien juottamista on hyvä asentaa ohjain koteloon, sillä useimmat napit kiinnitetään alapuolelta. Koska olen muovin työstämisessä vielä vähemmän auktoriteetti kuin juottamisessa, annan tähän vain ylimalkaiset ohjeet.

Irrota ohjaimen pallo, poraa varrelle noin 12 mm:n reikä ja pujota varsi läpi siitä. Kiinnitä pallo takaisin ja tee reiät ruuveille, joilla ohjaimen jalusta kiinnitetään. Tee sitten napeille noin 30 mm:n reiät (koosta riippuen) ja kiristä ne paikalleen. Tee lopuksi johdolle pieni kolo ulkoseinämään. Pyöröviila on hyödyllinen työkalu reikien hienosäädössä, sähkökäyttöinen pieni hiomakone vielä hyödyllisempi.

Voit myös 3D-tulostaa kotelon. Esimerkiksi Thingiversestä löytyy muutamia valmiita malleja. Tässä käyttämäni laitekotelo on iso, ruma ja vaikea työstää, mutta tulin sellaisen jo ostaneeksi, joten jatkamme sen pohjalta.

Kun saat ohjaimen ja napit kiinnitettyä, vedä aiemmin napeille varattu maajohto niiden toiseen napaan ja juota se kiinni. Juota sitten nappeihin niille tarkoitetut signaalijohtimet. Kuvassa 11 olen kuorinut maajohdon ensimmäiseltä napilta alkaen, jotta se mahtuu rei'istä läpi. Tästä ei ole haittaa, koska muut johdot on eristetty.

Sulkiessasi kotelon huolehdi, että johtoon aiemmin tekemäsi solmu jää kotelon sisään. Se poistaa vedon kaapelista, jolloin juotokset eivät irtoa, vaikka ohjainta riiputtaisi sen johdosta.



Kuva 11. Napit juotettuina.

Kuva 12. Reaalielämän vastine ns. koodarivääreille.

Valmista tuli

Funktionalistinen peliohjainlootamme on nyt valmis. Jos haluat varmistua sähköisestä toiminnasta, käytä nastajärjestyksen mittaamiseen alussa neuvomaani kikkaa – nyt vain tarvitset juotostinan pätkät molempiin yleismittarin antureihin. Laita toinen anturi maanastaan ja toinen siihen, jota haluat testata. Etenkin kannattaa varmistaa, ettei ole vahingossa kytkenyt mitään jännitteellisiä napoja (MSX:ssä nasta 5, Amigassa 7).

Luonnollisesti voit panostaa ohjaimeen niin paljon kuin haluat. Osiksi voi valita laadukkaamman tikun ja napit, ja kotelon voi tehdä vaikkapa puusta. Toimintaperiaate on ihan sama. Ja jos et koe vielä olevasi valmis juottamaan, voi liitokset tehdä myös puristettavilla ns. abiko-liittimillä.

TEKin JÄSENYYS – TYÖELÄMÄN TOIMIVIN KÄYTTÖLIITTYMÄ

YT:t? Uusi duuni haussa? Paljonko palkkaa? Kaipaako CV:si kohennusta tai työsopimuksesi tarkistusta? Vai etsitkö urallasi uutta suuntaa? Kiinnostaako yrittäjyys? Lähdössä ulkomaille? Haluatko verkostoitua alasi ammattilaisten kanssa?

Liity jäseneksi nyt, saat kaiken tämän:

- Työttömyysturva IAET-kassasta
- Sinulle räätälöity palkkaneuvonta
- Oman alasi edunvalvonta
- Kattavat työsuhdelakipalvelut
- Henkilökohtainen uraneuvonta, uratapahtumia ja CV-klinikka
- Ammattilehdet: TEK-lehti ja esim. Tekniikka&Talous sekä MikroPC sähköisesti tai paperilla
- Oikeusturva- ja vastuuvakuutus, ryhmäetuvakuutukset
- Yrittäjäneuvonta
- Lukuisia jäsentapahtumia ja jäsenalennuksia

Tekniikan akateemiset TEK on tekniikan alan yliopistokoulutettujen etu- ja palvelujärjestö. Jäsenet ovat tekniikan, tietojenkäsittelyn, arkkitehtuurin tai luonnontieteiden yliopistotutkinnon suorittaneita ja vastaavissa tehtävissä työskenteleviä korkeakoulututkinnon suorittaneita sekä näiden alojen opiskelijoita. TEK tukee jäseniään työuralla, edistää ammattikunnan osaamista ja yhteisöllisyyttä sekä rakentaa kestävää hyvinvointiyhteiskuntaa. Noin 73 000 jäsenellään TEK on Akavan toiseksi suurin jäsenliitto.

Mikä on peli?

Pelisuunnittelun filosofiaa, osa 1

Tuoreessa juttusarjassa pohdimme pelikehityksen perusasioita käytännön esimerkkien kautta. Ensimmäisellä kerralla käsittelyssä on kaikkein pyhin: pelin ydinsilmukka.

Teksti: Sami Koistinen
Kuvat: Mitol Berschewsky

Mikä oikeastaan tekee pelistä pelin? Määritelmiä on monia, mutta tässä eräs toimiva: peli antaa pelaajalle sekä haasteen että työkalut sen selvittämiseksi.

Monien tasohyppelypelien idea on yksinkertaisesti haastaa pelaaja kulkemaan pelikentän läpi. Mukaan voidaan lisämausteeksi laittaa vaarallisia vihollisia, jotka katkaisevat pelaajan etenemisen. Joskus kentästä selviytymistä vaikeuttaa aikaraja tai vaikkapa yksinkertaiset puzzle-elementit.

Pelimekaniikat muodostavat työkalupakin, joka on annettu pelaajalle pelikentän läpäisemiseksi. Tasoloikkapelien työkalupakkiin kuuluvat pelihahmon liikuttelu, hyppy ja kasa muita mahdollisia pelimekaniikkoja.

Kaikki tapahtuu pelin määrittelemien sääntöjen puitteissa. Säännöt voivat olla vaikkapa seuraavanlaisia: ”pelaaja kuolee osuttuaan viholliseen”, ”pelaajaan ja viholliseen vaikuttaa painovoima”, ”kenttää ei pääse läpi, jos aika loppuu” ja niin edelleen.

Haaste on vapaa

Jotkin pelit antavat pelaajan itse keksiä omat haasteensa. Myös tällaisissa hiekkalaatikkopeleissä on tarkkaan määritellyt työkalut ja säännöt, joiden puitteissa tavoitellaan itselle asetettuja haasteita.

Esimerkki vapaasti määriteltävän haasteen pelistä on Simcity, jossa pelaaja yrittää rakentaa hienon toimivan kaupungin, ajautumatta konkurssiin. Peli ei määrittele, millainen kaupungin tulee olla. Pelaaja saa päättää tämän aivan itse ja sitten yrittää parhaansa.

Myös Skyrim antaa pelaajalle vapaat kädet toteuttaa unelmansa – pelin sallimissa rajoissa. Moni suorittaakin vain muutamia tehtäviä, kunnes uusi salainen virtuaalielämä alkaa. Skyrimissa voi ostaa taloja, mennä naimisiin, käydä kauppaa, valmistaa esineitä ja tehdä paljon muutakin arkipäiväistä. Ja kaikki tämä ilman, että pelaaja olisi pakotettu läpäisemään peli. Kuitenkin nämä arkisetkin puuhat vaativat pelaajalta paljon työtä ja suunnitelmallisuutta.

Pelejä kaikki tyynni

Käyttämämme pelin määritelmä – peli antaa pelaajalle sekä haasteen että työkalut sen selvittämiseksi – ei rajoitu digitaaliseen universumiin. Esimerkiksi pokeri ja erilaiset urheilulajit joukkuepeleistä yksilösuorituksiin ovat selkeästi määritelmän sisällä.

Myös monet elämänvalinnat voidaan nähdä saman määritelmän kautta. Vaikkapa pörssimeklarin, urapyrkyrin tai poliitikon tietä kohti vaikutusvaltaa voidaan pitää pelinä – siis joukkona haasteita, jotka on ylitettävä.

Kun pelin sääntöinä ja pelikenttänä on todellinen maailma, ollaan niin sanotun ironman-pelin parissa. Käytössä on vain yksi elämä, eikä epäonnistumisen jälkeen voi ladata aiemmin tallennettua

tilannetta – eli juuri niin kuin Nethackissa.

Ytimessä silmukka

Kaikki pelit sisältävät ydinsilmukan, vaikka joskus sitä voikin olla hankalaa tunnistaa valmiista pelistä. Kaikessa yksinkertaisuudessaan silmukka on vain lista toimintoja, joita pelaaja suorittaa päästäkseen lähemmäksi pelin tavoitetta.

Pelit sisältävät usein erilaisia resursseja, joita pelaaja voi käyttää pelin edistämiseen. Niitä voivat olla esimerkiksi raha, aseet, lisäaika ja monet muut asiat. Usein resurssit ovat olennainen osa pelin ydinsilmukkaa.

Hyvä esimerkki tällaisesta silmukasta löytyy legendaarisesta avaruus- shoot'em up -pelistä Gradius, joka tunnetaan joillakin laitealustoilla nimellä Nemesis. Gradiuksessa pelaaja lentää avaruudessa ja ampuu vihollisia. Jotkin tuhotut viholliset jättävät jälkeensä power-up-resurssin eli esineen, jonka voi kerätä talteen. Sen poimittuaan pelaaja pystyy päivittämään aluksensa aseet parempiin. (Kuva 1.)

> *Peli antaa pelaajalle haasteen sekä työkalut sen ylittämiseksi.*

Pelaajat suorittavat Gradiuksen silmukan tehtävälistaa järjestyksessä, parhaansa mukaan. Pelitaidot määräävät, kuinka hyvin silmukan eri kohdista suoriudutaan.

Gradiuksen ydinsilmukka pyörii hyvin lähellä matalan tason pelimekaniikkoja. Resurssien keräys ja käyttö tapahtuvat aina pelikentällä, keskellä taistelun tiimellystä. Tämä auttaa pelaajaa selviytymään hengissä kentän läpi. Toisaalta ydinsilmukan voi rakentaa myös niin, ettei se ole sidottu näin kiinteästi matalan tason pelimekaniikkaan.

Korkeammalle tasolle

Ydinsilmukan mekaniikat ovat pelin sydän. Ilman niitä peliä ei voi pelata. Ydinmekaniikat ovat luuranko, jonka päälle ripustetaan muita, vähemmän tärkeitä toimintoja.

Ydinmekaniikan tunnusmerkki on, että pelaaja joutuu käyttämään sitä jatkuvasti pelin eri vaiheista selvitäkseen. Jalkapallossa esimerkkejä tästä ovat juokseminen ja pallon potkaiseminen. Sen sijaan vastustajan kampittaminen ei ole osa ydinmekaniikkaa, vaan sen ympärillä tapahtuva vapaaehtoinen, epäurheilijamainen toiminto.

Kaikissa peleissä ydinsilmukkaa ei suoriteta varsinaisen pelikentän sisällä. Joskus pelikenttä ja sen tapahtumat ovat vain yksi silmukan osa ja itse silmukka pyörii metapelitasolla. Hyviä esimerkkejä tällaisesta ovat vaikkapa Mass Effect -sarjan pelit, joissa pelihahmot saavat tehtävän suorittamisesta palkkioksi pisteitä, joita voi sijoittaa taitoihin.

Mass Effect -pelien ydinpelimekaniikkoja ovat juokseminen, aseella tähtääminen, ampuminen, erikoistaitojen käyttö, tavaroiden keräys sekä liuta muita ominaisuuksia. Nämä kaikki tapahtuvat pelikentän sisällä. (Kuva 2.)

On vaikeampaa sanoa, ovatko esimerkiksi keskustelun aloitus, keskusteluvalinnat, hahmon taitojen parantelu tai vaikkapa tähtijärjestelmien välillä liikkuminen ydinmekaniikkoja. Yleisesti voidaan kuitenkin sanoa, että mitä paremmin pelaaja hallitsee ydinpelimekaniikat, sitä paremmin hän suoriutuu pelin asettamista haasteista.

Kuva 1. Gradiuksen ydinsilmukka koostuu seuraavista toimenpiteistä:
1. Tuhoa kokonainen vihollisketju tai tietynvärinen vihollinen.
2. Kerää vihollisten jättämät resurssit avaruudesta.
3. Päivitä resurssien avulla omat aseesi paremmiksi, jotta saat tuhottua vihollisia helpommin.

Kuva 2. Mass Effectin ydinsilmukka koostuu seuraavista toimenpiteistä:
1. Suorita tehtävä, niin saat kokemuspisteitä taitoihin sekä uutta tavaraa.
2. Päivitä keräämiesi resurssien avulla pelihahmot ja varusteet. Tämä helpottaa tulevien tehtävien suorittamista.
3. Etsi uusi tehtävä, ja pelin tarina jatkuu.

Jalkapallo pelinä

Käytännössä kaikki pelit sisältävät seuraavat asiat:

1. Säännöt, jotka määrittelevät pelin raamit, joiden puitteissa kaikki tapahtuu.
2. Tavoite, joka pelaajan täytyy saavuttaa sääntöjen puitteissa.
3. Haaste, joka vaikeuttaa tavoitteen saavuttamista ja tekee pelistä mielenkiintoisen.
4. Metapeli, joka on pelin etenemisen mittari.
5. Palaute, josta pelaaja tietää, mitä pelissä tapahtuu.

Perinteinen jalkapallo voidaan määritellä yllä olevilla käsitteillä – sääntöjä hieman yksinkertaistaen – vaikkapa seuraavasti.

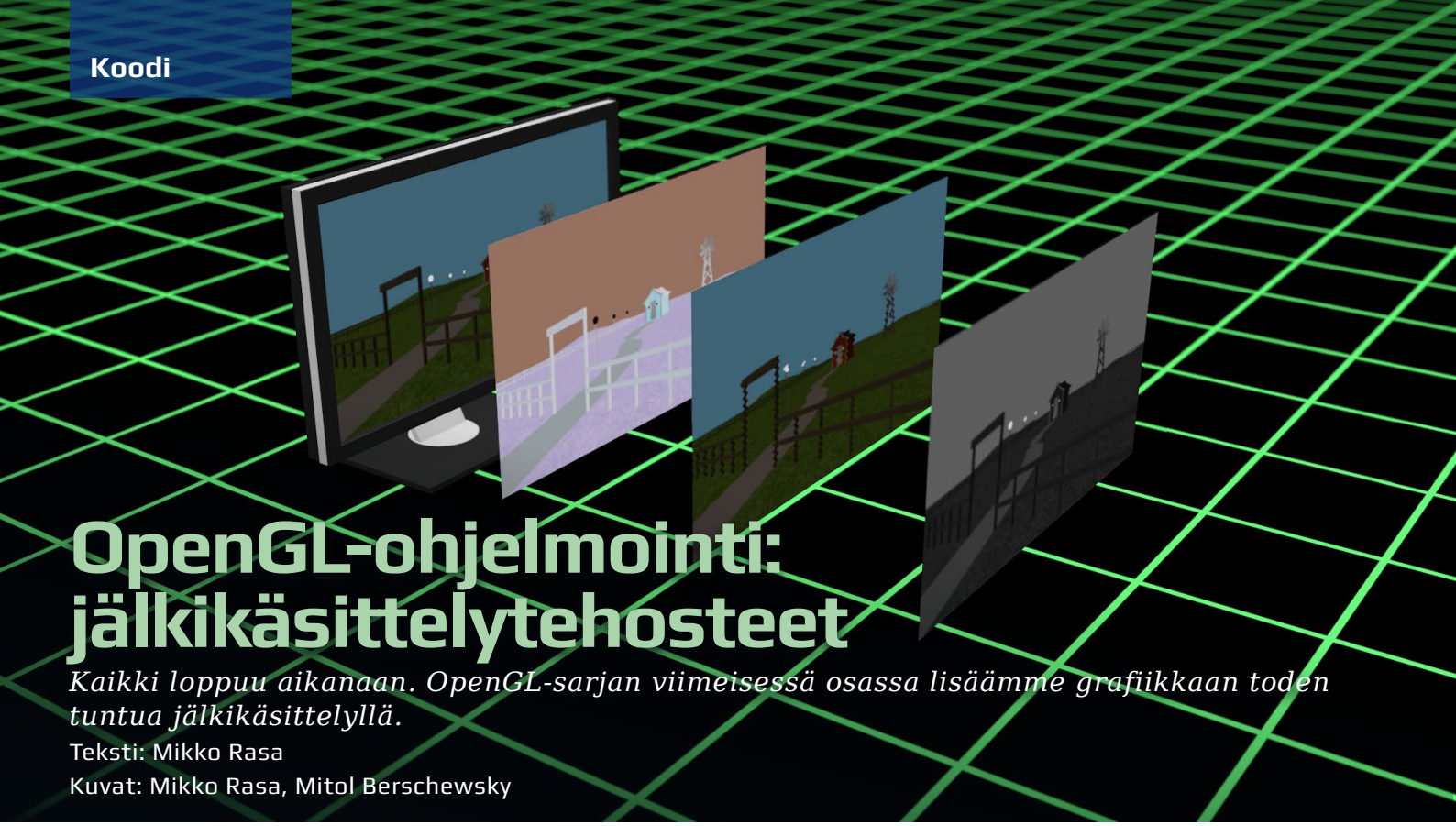
Säännöt: Pelikentällä on pallo. Palloon ei saa koskea käsillä. Voit liikuttaa palloa muilla ruumiinosillasi. Joukkueessasi on muitakin pelaajia, jotka saavat liikuttaa palloa samoilla säännöillä kuin sinä.

Tavoite: Liikuta pallo maaliin.

Haaste: Kentällä on toinenkin joukkue, jolla on samat säännöt, sama pallo ja sama tavoite mutta eri maali. Maalin edessä on vahti, joka yrittää estää palloa menemästä maaliin. Pallo ei saa poistua kentältä.

Metapeli: Jokainen maaliin saatu pallo antaa joukkueelle yhden pisteen. Kun aikaa on kulunut 90 minuuttia, eniten pisteitä kerännyt joukkue voittaa.

Palaute: Mitä lähempänä pallo on vastustajan maalia, sitä parempi tilanne on. Mitä lähempänä pallo on oman joukkueen maalia, sitä huonompi tilanne on. Mitä enemmän sinulla on pisteitä vastustajaasi nähden, sitä parempi on tilanne. Jos erotuomari näyttää sinulle punaista korttia, menit soheltamaan kentällä pahasti.



OpenGL-ohjelmointi: jälkikäsittelytehosteet

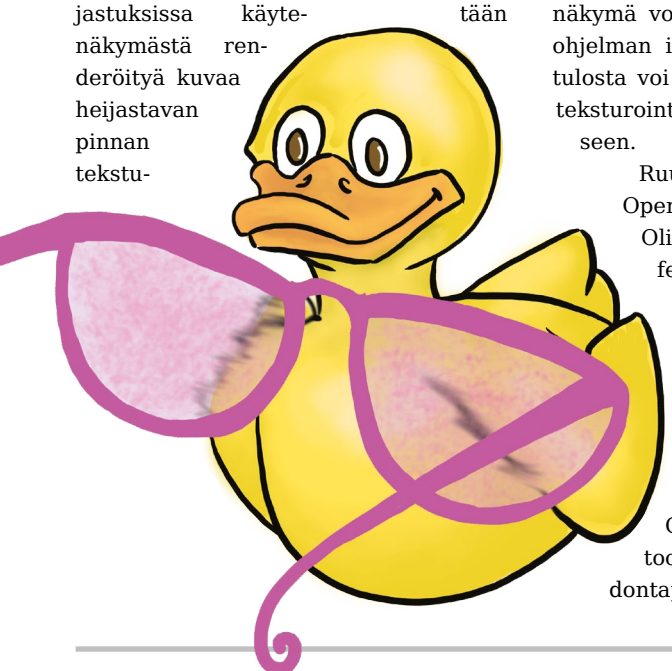
Kaikki loppuu aikanaan. OpenGL-sarjan viimeisessä osassa lisäämme grafiikkaan toden tuntua jälkikäsittelyllä.

Teksti: Mikko Rasa
Kuvat: Mikko Rasa, Mitol Berschewsky

Yksityiskohtaiset mallit ja tarkat tekstuurit ovat 3D-grafiikassa tärkeitä, mutta viimeinen silaus luodaan jälkikäsittelytehosteilla. Nimensä mukaisesti tehosteilla käsitellään kuvaa sen jälkeen, kun kaikki esineet on piirretty.

Mahdollisuudet ovat rajattomat. Yleisiä jälkikäsittelynä toteutettavia tehosteita ovat esimerkiksi liikepehmennys, hehku, ympäristön varjostus ja sRGB-värimuunnos. Jopa valaistuslaskenta on mahdollista siirtää kokonaan jälkikäsittelyyn, jolloin vältetään turhaa laskentaa pikseleille, jotka jäävät piiloon.

Samoja tekniikoita voi käyttää myös muihin tarkoituksiin. Esimerkiksi heijastuksissa käytetään näkymästä renderöityä kuvaa heijastavan pinnan tekstu-



rointiin. Jos tekstuuri on peräisin eri näkymästä, voidaan toteuttaa vaikkapa valvontakameran kuva vartiointihuoneen monitorissa tai maaginen portaali toiseen ulottuvuuteen.

Tässä artikkelissa toteutamme hehkutehosteen. Se parantaa realismia sellaisissa näkymissä, jotka sisältävät erittäin kirkkaita alueita. Pohjana käytämme viime Skrollissa esiteltyä grafiikkamoottoria.

Ruutupuskurioliot

Ennen kuin käymme itse tehosteen kimppuun, tutustumme yhteen uuteen OpenGL:n käsitteeseen: ruutupuskurioliioihin (framebuffer objects). Niiden avulla näkymä voidaan renderöidä tekstuuriin ohjelman ikkunan sijaan. Renderöinnin tulosta voi sen jälkeen käyttää pintojen teksturointiin tai tehosteiden toteuttamiseen.

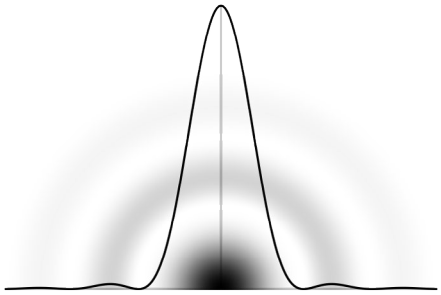
Ruutupuskurioliot noudattavat OpenGL:n yleistä oliorajapintaa. Oliotyypin nimi on Framebuffer. Sidontapisteitä on kaksi: GL_DRAW_FRAMEBUFFER toimii piirtokomentojen kohteena ja GL_READ_FRAMEBUFFER lukukomentojen lähteenä. Useimmissa tapauksissa voi myös käyttää yhdistettyä sidontapistettä GL_FRAMEBUFFER, joka sitoo saman olion molempiin sidontapisteisiin.

Ruutupuskuriolio on säiliö eikä sellaisenaan kovin hyödyllinen. Siihen on liitettävä ainakin yksi kohdepuskuri ennen kuin sitä voi käyttää renderöintiin. Kohdepuskurit voivat olla joko tekstuureja tai erityisiä renderöintipuskureita, jotka käsittelemme hieman jäljempänä. Liitettävissä puskureissa voi olla väri-, syvyys- tai sapluuna-arvoja.

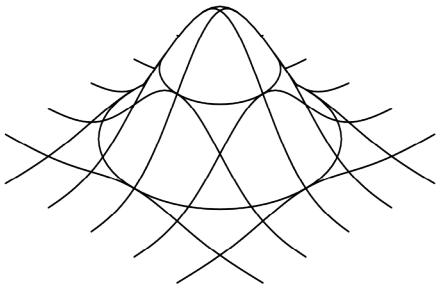
Kaksiulotteinen tekstuuri liitetään ruutupuskuriolioon komennolla glFramebufferTexture2D. Toiminto kohdistuu sidottuun ruutupuskuriin, mutta tekstuurin ei tarvitse olla sidottu. Tekstuurille ei tarvitse olla varattu muistia liittämisen aikana, vaan riittää, kun se varataan ennen ruutupuskuriolion käyttöä. Renderöinnin kohteena olevalle tekstuurille kannattaa tehdä pelkkä muistinvaraus, jolloin ei tarvitse turhaan kopioida pikselidataa ohjelman osoiteavaruudesta näytönohjaimelle.

Ruutupuskuriolioihin voi liittää myös yksiulotteisia tekstuureja, kolmiulotteisten tekstuurien viipaleita tai kuutiotekstuurien sivuja. Niitä kuitenkin tarvitaan harvemmin, joten emme käy tässä läpi yksityiskohtia.

Monissa tapauksissa tehosteen toteuttaminen vaatii vain väripuskuria, mutta näkymän renderöintiin tarvitaan myös syvyyspuskuri. Tällöin voidaan käyttää renderöintipuskureita (renderbuffer objects) toissijaisten osapuskurien kohteena. Renderöintipuskureita käytetään vain tähän tarkoitukseen, joten



Airyn diffraktiokuviossa on keskellä olevan kiekon ympärillä renkaita, joiden kirkkaus heikkenee etäisyyden kasvaessa. Kuvassa renkaita on voimistettu erottuvuuden parantamiseksi.



Gaussilaisen pehmennyksen toteuttavan konvoluutioytimen kaikki poikkileikkaukset ovat Gaussin käyriä ja korkeuskäyrät ovat ympyröitä.

> Konvoluutiota voidaan soveltaa signaalinkäsittelyssä myös yleisemmin. Sitä ei ole rajattu kuviin, vaan yhtä hyvin sitä voi käyttää myös ääneen tai radiosignaaliin. Signaalin ei tarvitse edes olla diskreetti eli näytepisteistä koostuva. Analogisilla piireillä voidaan suorittaa konvoluutiota jatkuville signaaleille.

toteutuksella on enemmän vapauksia tehokkaan säilytystavan ja käsittelyalgoritmien valintaan.

Renderöintipuskurien oliotyypin nimi on Renderbuffer, ja tarjolla on vain yksi sidontapiste, GL_RENDERBUFFER. Renderöintipuskurin koko ja formaatti asetetaan komennolla glRenderbufferStorage. Tämän voi tehdä vain kerran puskurin elinaikana.

Ruutupuskuriolion osapuskurit voivat olla erikokoisia kuin järjestelmän tarjoama ruutupuskuri ja myös keskenään erikokoisia. OpenGL-toteutus ei voi tietää, mitä osaa halutaan käyttää, joten kuva-alue on määriteltävä komennolla glViewport. Toisin kuin joissakin muissa grafiikkarajapinnoissa, ruutukoordinaattien nollapiste on vasemmassa alakulmassa ja Y-koordinaatti kasvaa ylöspäin. Kuva-alue on globaalia tilaa, joten se on parasta asettaa uudestaan aina kun käytössä oleva ruutupuskuri vaihtuu.

Ollakseen käyttökelpoinen ruutupuskuriolion on oltava eheä. Sillä on seuraavanlaisia vaatimuksia: jokaisen osapuskurin täytyy olla eheä, piirtokomentojen kohteena olevan väripuskurin (yleensä GL_COLOR_ATTACHMENT0) pitää olla liitetty ja kaikkien osapuskurien moninäytteistysasetusten on oltava samat. Lisäksi OpenGL-toteutus voi asettaa rajoituksia osapuskureissa käytettyjen formaattien yhdistelmälle. Ruutupuskuriolion eheyden voi tarkistaa komennolla glCheckFramebufferStatus.

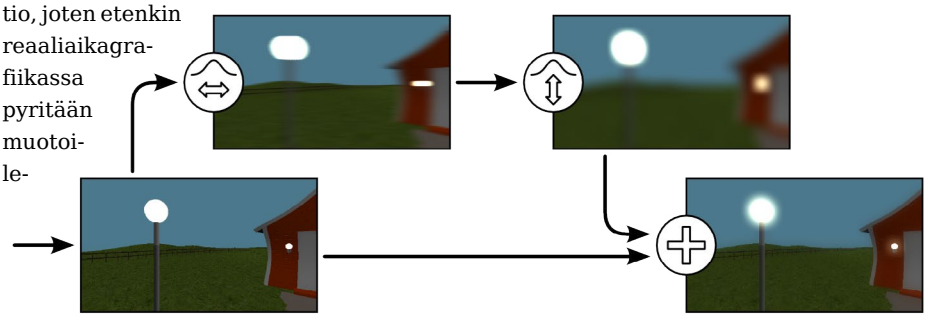
Jos tekstuuri on liitetty käytössä olevaan ruutupuskuriolioon, sitä ei voi samanaikaisesti käyttää teksturointiin. Tulos on OpenGL-standardin mukaan määrittelemätön.

Hehkutehosteen teoria

Pohjimmiltaan hehkutehosteessa on kyse optiikan epätäydellisyydestä. Virheetönkin linssi tuottaa diffraktiokuvion, jota kutsutaan Airyn kiekoksi. Sen ansiosta kuvassa olevat kirkkaat alueet luovat ympärilleen hehkun, joka leviää ympäröiviin tummiin alueisiin. Tällaisia tapauksia ovat esimerkiksi auringonvaloa tulvivan ikkunan edessä oleva siluetti, pimeydestä lähestyvän auton ajovalot ja seikkailijan kantama soihtu pimeässä luolassa.

Operaatiota, jossa kukin pikseli vaikuttaa ympäristöönsä, kutsutaan konvoluutioksi (engl. convolution). Pikselien vaikutukset annetaan yleensä matriisina, jota kutsutaan konvoluution ytimeksi (engl. kernel). Kaksiulotteisen konvoluution laskeminen on hyvin raskas operaatio, joten etenkin reaaliaikagrafiikassa pyritään muotoilemaan ytimet niin, että ne voidaan esittää vaaka- ja pystysuuntaisen konvoluution yhdistelmänä. Tällaista ydintä kutsutaan separoituvaksi.

Valitettavasti Airyn kiekko ei ole separoituva. Sen sijaan Gaussin käyrään perustuva gaussilainen pehmennys (engl. gaussian blur) on. Sopivalla skaalauksella päästään hyvin lähelle oikeaa lopputulosta. Fysikaalisesti oikein tehty tehoste on kuitenkin useimmissa tapauksissa lähes huomaamaton, joten usein tehostetta korostetaan levittämällä pehmennys laajemmalle alueelle. Tällöin pehmennetty kuva kannattaa sekoittaa alkuperäiseen kontrastien säilyttämiseksi.



Hehkutehosteen vaiheet.

OpenGL:n ruutupuskuritajapinta

Funktiot:

```
void glGenFramebuffers(GLsizei n, GLuint *framebuffers);
void glBindFramebuffer(GLenum target, GLuint framebuffer);
void glDeleteFramebuffers(GLsizei n, const GLuint *framebuffers);
void glFramebufferTexture2D(GLenum target, GLenum attachment,
        GLenum textarget, GLuint texture, GLint level);
void glFramebufferRenderbuffer(GLenum target, GLenum attachment,
        GLenum renderbuffertarget, GLuint renderbuffer);
GLenum glCheckFramebufferStatus(GLenum target);
void glBindRenderbuffer(GLenum target, GLuint renderbuffer);
void glDeleteRenderbuffers(GLsizei n, const GLuint *renderbuffers);
void glGenRenderbuffers(GLsizei n, GLuint *renderbuffers);
void glRenderbufferStorage(GLenum target, GLenum internalformat, GLsizei width,
        GLsizei height);
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Sidontapisteet:

```
GL_DRAW_FRAMEBUFFER
GL_READ_FRAMEBUFFER
GL_FRAMEBUFFER
```

Liitospisteet:

```
GL_COLOR_ATTACHMENTi
        i on nollasta alkava kokonaisluku
GL_DEPTH_ATTACHMENT
GL_STENCIL_ATTACHMENT
```

Hehkutehosteen toteutus

Jälkikäsittelytehosteet piirtävät yleensä yhden suorakulmion, joka täyttää koko ruudun. Siihen käytettävä kulmapistehshaderi on hyvin yksinkertainen. Niinpä sivuutamme sen ja keskitymme pikselishadereihin, joissa varsinainen tehoste toteutetaan.

Hehkutehosteen aikaansaamiseen tarvitaan kolme vaihetta. Ensimmäisessä vaiheessa suoritetaan vaakasuuntainen pehmennys ja toisessa pystysuuntainen. Kolmannessa vaiheessa yhdistetään pehmennetty kuva alkuperäiseen. Kaksi ensimmäistä vaihetta voidaan toteuttaa samalla shaderilla, joka näyttää seuraavalta:

```
uniform sampler2D source;
uniform vec2 delta;
uniform float factors[19];
in vec2 texcoord;
out vec4 out_color;
void main()
{
  out_color = vec4(0.0, 0.0, 0.0, 0.0);
  for(int i=0; i<19; ++i)
  out_color += texture2D(source,
      texcoord+delta*float(i-9)
    )*factors[i];
}
```

Gaussilaisen pehmennyksen toteuttava pikselishaderi.

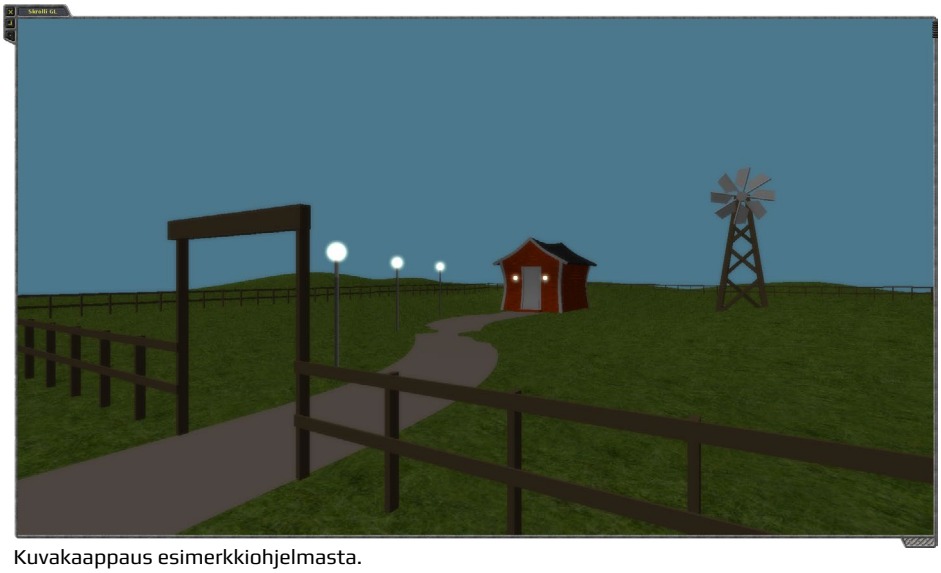
Uniform-muuttuja source sisältää lähdetekstuurin. Ensimmäisessä vaiheessa tämä on näkymästä renderöity kuva ja toisessa vaiheessa ensimmäisen vaiheen tulos.

Delta on kahden näytepisteen välinen etäisyys tekstuurikoordinaateissa. Se on muotoa (x, 0) tai (0, y) riippuen siitä, kumpaan suuntaan pehmennys tapahtuu.

Factors sisältää konvoluutioytimen painoarvot. Ytimen on hyvä olla normalisoitu, eli arvojen summan pitäisi olla 1. Taulukon koon on oltava pariton, jotta näytteitä on yhtä monta keskipisteen molemmin puolin. Jos sen kokoa muutetaan, on muutettava myös silmukassa olevia arvoja vastaavasti.

Ytimen koon ja näytepisteiden välisen etäisyyden valinta on kompromissi laadun, vaikuttavuuden ja suorituskyvyn välillä. Kasvattamalla etäisyyttä pehmennyksen laajuus kasvaa ja tehosteesta tulee voimakkaampi. Samalla kuitenkin kuvaan syntyy artefakteja pienten yksityiskohtien ympärille. Ytimen kokoa kasvattamalla saadaan vastaava lopputulos ilman artefakteja, mutta näytteitä joudutaan ottamaan enemmän, ja suorituskyky huononee.

Shaderin toiminnallinen osuus laskee näytteistä summan, jossa käytetään konvoluutioydintä painoarvoina. Tekstuurikoordinaatteja laskettaessa silmukkalaskurista vähennetään puolet ytimen koosta, jolloin näytteet sijoittuvat tasaisesti käsiteltävän pikselin molemmin puolin.



Kuvakaappaus esimerkkiohjelmasta.

```
uniform sampler2D source;
uniform sampler2D blurred;
uniform float strength;
in vec2 texcoord;
out vec4 out_color;
void main()
{
    out_color = mix(
        texture2D(source, texcoord),
        texture2D(blurred, texcoord),
        strength
    );
}
```

Kuvat yhdistävä pikselishaderi.

Uniform-muuttuja source sisältää alkuperäisen kuvan näkymästä, blurred puolestaan pehmennetyn version.

Strength on sekoitussuhde, jolla alkuperäinen ja pehmennetty versio yhdistetään. Jos sen arvo on 0, käytetään pelkästään alkuperäistä kuvaa. Arvolla 1 käytetään pelkästään pehmennettyä kuvaa. Tälläkin voi vaikuttaa tehosteen voimakkuuteen.

Shaderi käyttää sekoitukseen GLSL:n mix-funktiota. Se laskee lineaarisen interpolaation kahden ensimmäisen parametrin välillä käyttäen kolmatta parametria interpolaatiomuuttujana.

Tehoste on myös yhdistettävä muuhun grafiikkamoottoriin jotenkin. Tätä varten teemme uuden luokkahierarkian, jonka kantaluokka saa nimekseen PostprocessingEffect. Mikäli tehosteita on samaan aikaan käytössä useampia, on edellisen tehosteen lopputulos annettava syötteeksi seuraavalle.

Koska hehkutehosteessa on kyse suurista kirkkauksista, ei tavanomaisten 8-bittisten värien dynamiikka riitä. Parempi onkin asettaa kohdetekstuurille liukulukuformaatti, kuten GL_RGB16F. Tällöin pikselishaderin tuottamat väriarvot tallennetaan sellaisenaan eikä niitä rajoiteta 0:n ja 1:n välille.

Esimerkkitoteutus on saatavissa Skrollin verkkosivuilta osoitteesta http://www.skrolli.fi/2014.2/.

Kirjoittaja vastaa mielellään OpenGL:ää koskeviin kysymyksiin Ircnetissä nimimerkillä tdb, kanavilla #skrolli ja #opengl.fi.

Koululaisen pieni peliohjelmointiopas

Viime vuosina on ilahduttavasti heräilty siihen, että ohjelmointitaito on tärkeää ja että sitä on hyvä opettaa etenkin lapsille ja nuorille. Opetushankkeet ovat ruvenneet poikimaan myös oppikirjoja.

Teksti: Ville-Matias Heikkilä

Koululaisen pieni peliohjelmointiopas syntyi osana Kajaanin ammattikorkeakoulun OHTA-hanketta (Ohjelmointi tavaksi), jossa pidettiin ohjelmointikerhoa alueen peruskoululaisille vuosina 2011–2013. Kerhot suunnattiin erityisesti 10–15-vuotiaille ja kielenä käytettiin Microsoftin Small Basicia.

80-sivuisen kirjan ensimmäisessä puoliskossa paneudutaan kielen ja ohjelmointiympäristön alkeisiin ihan sen asentamisesta alkaen. Lähestymistapa muistuttaa hyvin paljon 1980-luvun mikrotietokonekirjoja aina esimerkkiohjelmia myöten: muuttujien toimintaa harjoitellaan ”Mikä on nimesi?” -ohjelmalla, silmukoita puolestaan ”raketin lähtölaskennalla”. Tekijät ovat selvästikin muistelleet omia alkuaikojaan ja ottaneet niistä vaikutteita. Myös Basicin valintaa kieleksi voidaan pitää varsin perinnetietoisena, vaikkei Small Basic kovin lähellä vanhan ajan basiceja olekaan.

Kirjan jälkipuoliskossa keskitytään kahden laajemman peliohjelman rakentamiseen. Ensimmäinen näistä on tekstipohjainen seikkailupeli, jälkimmäinen puolestaan graafinen autopeli. Molempien listaukset on painettu kirjaan kokonaisuudessaan, mutta ne ovat sen verran pitkiä, että useimmat päätynevät mieluummin lataamaan ne OHTA-projektin sivustolta. Molemmissa peleissä myös käytetään sellaisia kielen ominaisuuksia, joita ei käsitellä kirjan alkupuolella lainkaan. Opas näyttäisi siis puolenvälin tienoilla muuttuvan äkkiä paljon vaativammaksi.

Osa kirjan sisällöstä tuntuu jotenkin irralliselta, esimerkiksi aukeaman mittainen osio tekstiseikkailujen historiasta tai lista Small Basicin tuntemista matematiikkafunktioista. Eikö jälkimmäisen sijaan olisi ollut hyvä esitellä vaikkapa taulukkotietorakenne, jota molemmat esimerkkipelit hyödyntävät? Ehkä tilaa olisi voinut käyttää myös muutaman rivin mittaisiin esimerkkiohjelmiin, joissa esitellään mielenkiintoisia graafisia ja muita temppuja. Muistelen itse noin kahdeksanvuotiaana selailleeni ohjelmointikirjoja ja lehtien listauksia etenkin siinä toivossa, että niistä löytyy mielenkiintoisia niksejä.

Ohjelmointiympäristön valintaa voi myös kritisoida. Small Basic on saatavilla vain Windowsille, jota yhä useamman henkilökohtaisina tietokoneinaan käyttämät tabletit ja puhelimet eivät aja. Mahdollisuus tehdä kavereiden laitteissa toimivia pelejä antaisi paljon paremmat eväät uudelle nuorison ohjelmointikulttuurille.

Muutamista ongelmistaan huolimatta Koululaisen pienen peliohjelmointioppaan kaltaiset kirjat ovat hyvin tervetulleita nykyiseen tietokonekulttuuriin. Myös muita samansuuntaisia projekteja kannattaa toki pitää silmällä – esimerkiksi lapsille Ruby-kielellä ohjelmointia opettavaa Hello Ruby -kirjaa, jota työstetään paraikaa joukkorahoituksen turvin.

Opas testissä

Teksti: Toni Kuokkanen

Peliohjelmointiopas testattiin myös autenttisella yleisöllä: 9-, 11- ja 13-vuotiailla innokkailla. Testiyleisön mielestä asiat käytiin kirjassa läpi riittävän selkeästi ja kädestä pitäen. Tiimin nuorinkin pääsi mukavasti koodaamisen makuun ja kahlasi kirjan läpi ensimmäisenä kolmikosta.

Vanhemmat testaajat kritisoivat hiukan sitä, että kirjassa käytiin jotkin asiat läpi liian hitaasti. Tämän kritiikin kohteeksi joutuneiden osioiden yli hypittiinkin helposti. Myös pelimusiikista olisi kaivattu omaa osiota. Nopeimmat kerkesivät jo aloittamaan omia kokeilujaan aiheesta.

Sivusta seuraavan näkökulmasta esimerkkiautopelin grafiikan ja ominaisuuksien muutteleminen vaikutti olevan hauskinta. Koko ryhmä kävi melko nopeasti kirjan kokonaisuudessaan läpi, ja jatko-osaa ehdittiin jo kaipailla.

Jumala koneesta – TempleOS

Käyttöjärjestelmä on keskeinen osa jokapäiväistä tietokoneen käyttöä. Moni ei edes tiedosta käyttöjärjestelmänsä merkitystä, mutta Terry Davisille käyttöjärjestelmä on temppeli – kirjaimellisesti.

Teksti: Kalle Viiri

Kuvat: Manu Pärssinen, Kalle Viiri

TempleOS on omalaatuinen käyttöjärjestelmä, joka on kehitetty näpertelijän leikkikentäksi. Commodore 64:stä mallia ottaen se ei tunne tiedostojen oikeuksien rajoittamista tai ohjelmakoodin suoritustasojen erottelua. Kaikki koodi suoritetaan täysillä oikeuksilla ”kernel modessa”.

Erikoista käyttöjärjestelmässä ei kuitenkaan ole vain sen tietoisesti olemattomat suojaustoimet. TempleOS:n kehittäjän **Terry Davisin** mukaan se on nimensä mukaisesti Jumalalle pyhitetty temppeli, jossa rukoillaan, uhrataan ja joskus saadaan myös vastauksia rukouksiin.

Bittitemppeli tosiuskoville

Davis kertoo TempleOS:n sivuilla sen vaatimusten tulleen Jumalalta. Samaan tapaan kuin Jumala Vanhassa testamentissa antaa Nooalle mitat arkkiin, Davis kertoo Jumalan määränneen TempleOS:n resoluution, äänentoiston, ikkunamallin ja muut ominaisuudet pieniäkin yksityiskohtia myöten.

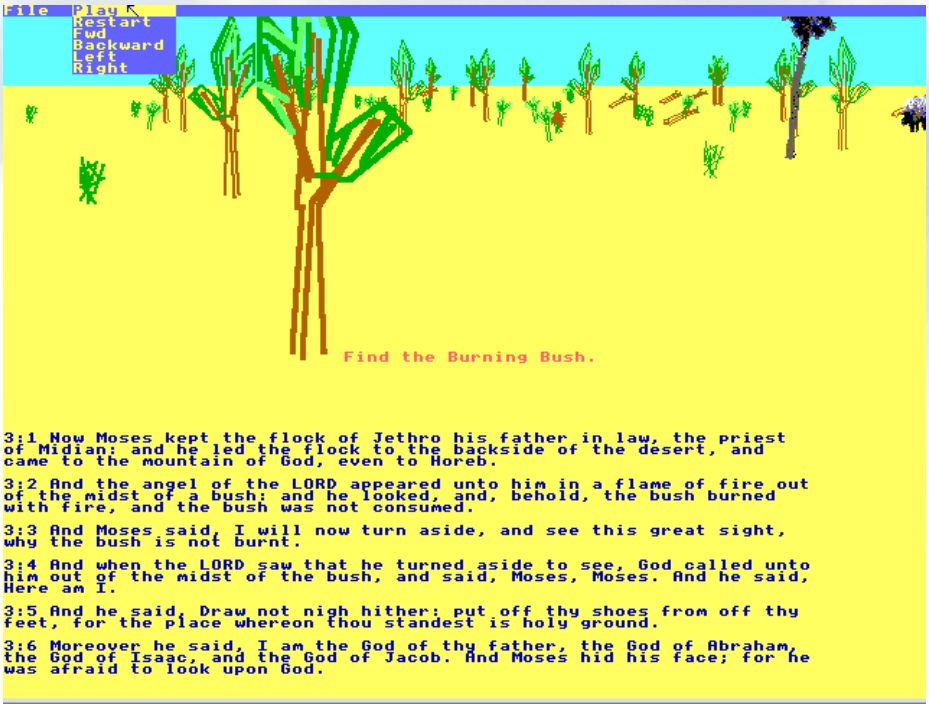
Davis mainitsee käyttöjärjestelmän sivuilla järjestelmän saaneen paljon inspiraatiota myös Commodore 64:stä, jolla itse aikanaan ohjelmoi paljon. Kaikki prosessit suoritetaan täysillä oikeuksilla, jotta harrastelijan ohjelmointi olisi mahdollisimman helppoa. TempleOS:n mukana tulee omia ohjelmointiprojekteja varten Davisin oma syntaktisella sokerilla makeutettu C++ -mukaelma HolyC, jota TempleOS:n komentotulkkikin tukee.

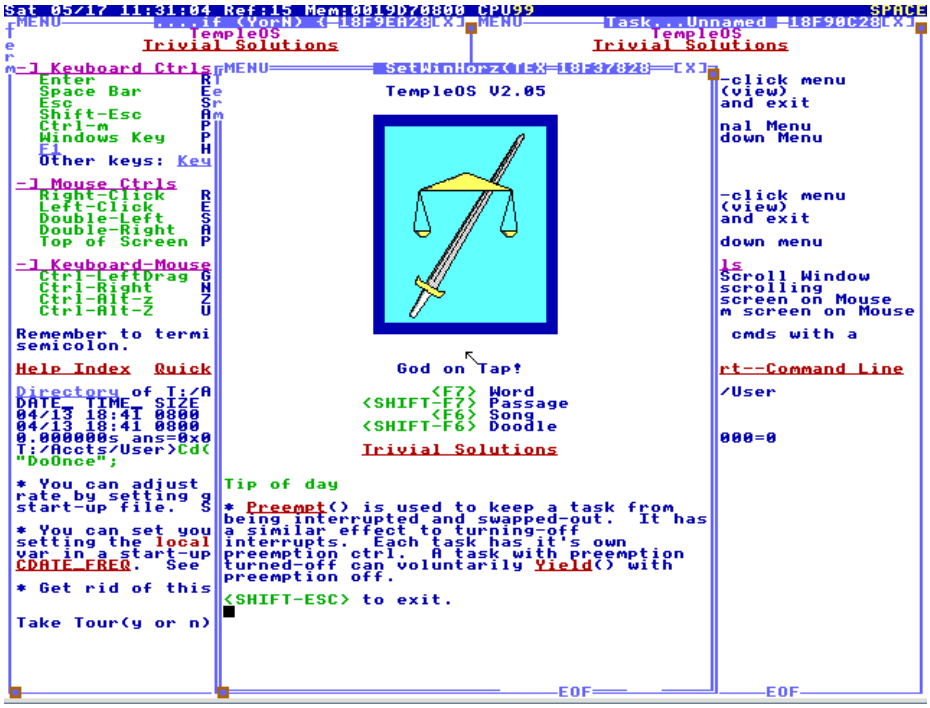
Rukouksesi on kuultu

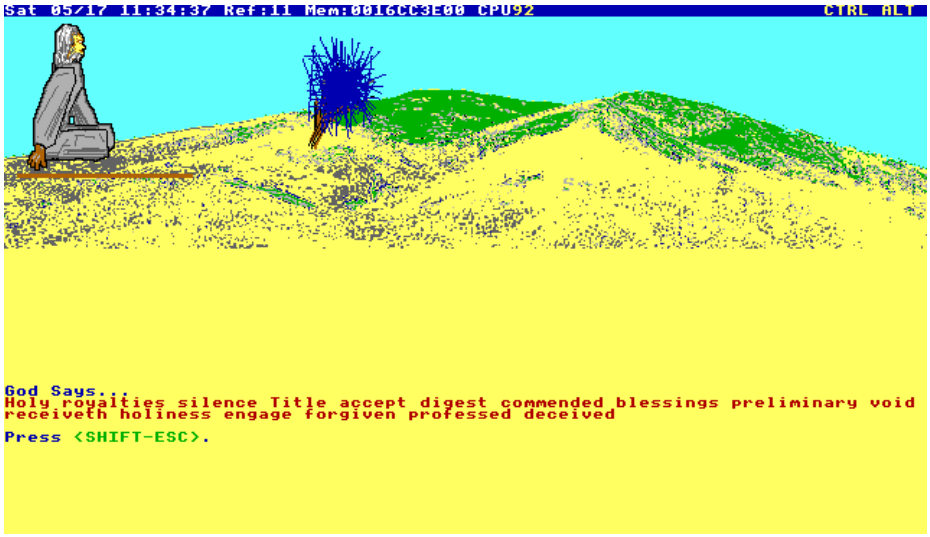
TempleOS:n keskeisimpiä toimintoja on oraakkeli, jolle käyttäjä voi uhrata. Davis itse uhraa sarjakuvia ja virsiä, joista osa tulee käyttöjärjestelmän mukana muidenkin nähtäviksi. Sarjakuvissa muutamasta viivasta koostuva Mooses-hahmo juttelee yllättävän kyynistä huumoria viljelevän Jumalan kanssa.

Virret piirtyvät ruudulle sanoina kutakuinkin taustalla soivan melodian tahtiin. Sekä virret että sarjakuvat on tehty TempleOS:n omilla työkaluilla, eli käyttäjä pääsee halutessaan itsekin kokeilemaan uhrilahjojen tekoa. Säveltämisen vaikeutta tosin lisää se, että ainakaan QEMU-virtuaalikone ei suostunut toistamaan ääniä.

Oraakkeli osaa myös vastata rukouksiin ja uhrilahjoihin valitsemalla Raamatusta otteita tai satunnaisia sanoja käyttäen tietokoneen kelloa satunnaislukugeneraattorina. Oraakkelin vastaus voi tuntua tottumattoman silmiin täysin järjettömältä, mutta Davisin mukaan kyseessä on ”kielillä puhuminen”, jonka





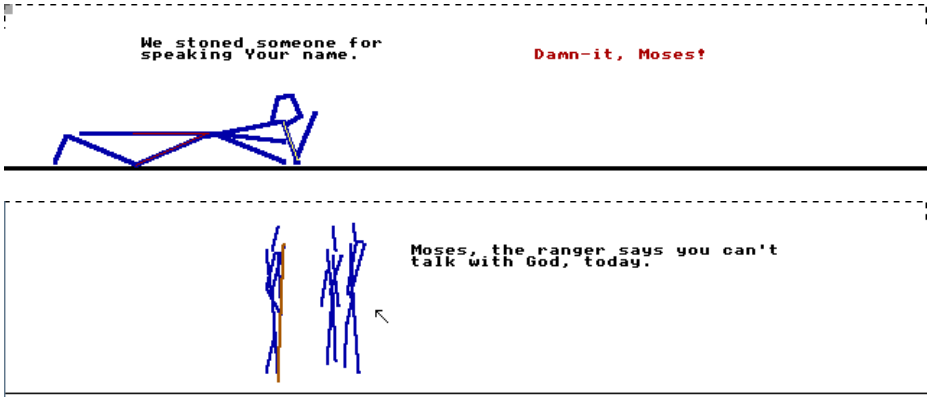


tulkitseminen ei ole täysin suoraviivaista.

Davis ei videoillaan peittele menetelmän yksinkertaisuutta tai yritä verhota sitä mystiikkaan. Hänen mukaansa Jumalan viestejä voi vastaanottaa niin vanhanaikaisilla Ouija-laudoilla kuin avaamalla kirjoja satunnaisista paikoista. Davis toteaa koneen kellon olevan vain modernimpi ja helpompi vastine näille.

Hartaita harrasteita

Oraakkeli ja sarjakuvat ovat osa After Egypt -ohjelmakokonaisuutta, jossa on mukana myös muutama Davisin ohjelmoima Raamatun tarinoihin perustuva peli tai demo. Pelit ovat yksinkertaisia otteita Israelin kansan vaiheista Egyptin orjuuden jälkeisen erämaavaelluksen varrelta. Peleinä on mutkattomia taistelusimulaattoreita, oikeudenkäyntiprosejoissa pelaaja saa valita armahtaako vai tuomitseeko rikollisia ja kiven iskemistä sauvalla veden saamiseksi.



After Egyptin lisäksi TempleOS:n mukana tulee myös muutama sekulaarisempi pelikokemus: pääpiirteittäin toimivia, mutta kehnoilla grafiikoilla varusteltuja ja tökkiviä kopioita tunnetummista peleistä. Tankkipeli Zone Out jopa julistaa alkuruudussaan olevansa tarkoituksella kehno kopio alkuperäisestä. ”Alkuperäinen” viitannee Battlezoneen.

Tie paratiisiin on kivinen

Kokonaisuutena TempleOS on erikoinen ja mielenkiintoinen kokemus, mutta varsinkin nykyajan mittapuulla kömpelöön järjestelmään on vaikea tottua ja tutustua. Järjestelmä ei myöskään pysynyt hyvin pystyssä QEMUlla kokeillessa, vaan kaatuili pahimmillaan muutaman minuutin välein. Davis myöntää, että TempleOS ei ole kovin turvallinen ja vakaa. Eikä sen tarvitsekaan olla, sillä lähtökohtaisesti sitä käytetään toisen käyttöjärjestelmän rinnalla. Hengellisen yhteydenkään muodostumisesta ei ole suoranaisia takeita. Ainakaan toimittaja ei kokenut valaistuneensa merkittävästi suuntaan tai toiseen.

Uskonto käyttöjärjestelmässä

Harva käyttöjärjestelmä väittää TempleOS:n tapaan olevansa suoraan korkeampien voimien tukema. Tavanomaisempia uskonnollisia käyttöjärjestelmiä sen sijaan on monia. Tunnetuimpia ja käytetyimpiä näistä on vapaaehtoisvoimin kehitetty **Sabily**, islaminuskoisille suunnattu versio Ubuntu GNU/Linux -jakelusta. Sabily lisää normaaliin Ubuntuun rukousaikakellon, Koraanin tutkimiseen tarkoitetun **zekr**-ohjelman, islamilaisen kalenterin, lapsilukon ja muita muslimeille hyödyllisiä ohjelmia. Kristityille suunnattu **Ubuntu Christian Edition** puolestaan sisältää Raamatun lukemiseen tarkoitettuja ohjelmistoja sekä Sabilyn tapaan lapsilukon. Sabilyn ja Christian Editionin perässä on seurannut joukko muita uskonnollisia Ubuntu-muunnelmia sekä parodioita.

Vähemmän vakava ajatus uskonnollisesta käyttöjärjestelmästä eli jo 1990-luvulla, jolloin Linuxista oltiin huhujen mukaan kehittämässä uskonnollista **Jesux**-versiota. Jesuxin ominaisuuksiin kuului mm. ”syntisen termistön” kuten sanojen **abort** ja **daemon** vähentäminen sekä sisäänkirjautumisen estäminen sunnuntaisin lepopäivän kunnioittamiseksi. Kyseessä oli kuitenkin vain huiputus, joka onnistui herättämään aikansa huomiota postituslistoilla.

Ei näin! SEGOILUA

Vuonna 1992 Segalla oli 65 prosenttia USA:n pelikonsolimarkkinoista. Alle kymmenen vuotta myöhemmin se luopui laitevalmistuksesta kokonaan. Miten tällainen temppu tehdään?

Teksti: Mikko Heinonen Kuvat: Wikimedia Commons -käyttäjä Evan-Amos

Segan esi-isä, yhtiö nimeltä Service Games, syntyi Havaijilla vuonna 1940. Toisen maailmansodan jälkeen perustajat Raymond Lemaire ja Richard Stewart siirsivät toiminnan Japaniin, jonne oli noussut paljon amerikkalaisten sotilastukikohtia. Service Games toi niihin emämaasta jukebokseja sun muuta viihdykettä. Yhdistyttyään kilpailijansa Rosen Enterprisesin kanssa yhtiö lyhensi nimensä Segaksi ja huomasi pian muuttuneensa erilaisten viihdelaitteiden maahantuojasta niiden valmistajaksi.

Kun pelihallibisnes kukoisti 1970-luvulla, Sega kasvoi sen mukana. Arcadepelien vauhti kuitenkin hyytyi 1980-luvun alussa ja liikevaihdon lasku pelästytti yhtiön amerikkalaiset omistajat. Määräysvalta alkoi siirtyä Japaniin. Samalla alettiin tähyillä kuluttajamarkkinoiden suuntaan, ja ensimmäinen pelikonsoli SG-1000 (Sega Game 1000) ilmestyi yhtä jalkaa Nintendo Famicomin kanssa vuonna 1983.

SG-1000 ei ollut menestys, mutta Sega jatkoi sen kehittelyä. Eri versioiden kautta siitä muodostui lopulta Sega Master Systeminä tunnettu konsoli. Vaikkei SMS:kään onnistunut Nintendon valta-asemaa horjuttamaan, se löysi kuitenkin omat ystävänsä etenkin Euroopasta ja Etelä-Amerikasta, eläen aina 90-luvun puolelle asti.

Jättivietti herää

Segan ainoa kiistaton menestystuote oli 16-bittinen Mega Drive (USA:ssa Genesis). Sen suunnittelu ammensi yhtiön parhaasta osaamisesta. Motorola MC68000:n ja Zilog Z80:n yhdistelmä oli jo aiemmin osoittautunut toimivaksi Segan System 16 -kolikkopeliraudassa, ja vuoteen 1989 mennessä sen hinta oli laskenut massatuotantoon sopivaksi.

Menestys etenkin Amerikassa haluttiinkin tällä kertaa varmistaa. Sega oli antanut kenkää markkinoinnista aiemmin vastanneelle leluyhtiö Tonkalle ja rakentanut Sega of American käytännössä tyhjästä, rekrytoimalla alan huippuosaajia. Ja koska eräs tämän juttusarjan perusteemoista on nauraa Atarin möhläilyille, todettakoon että Sega tarjosi Genesistä myös Atarin edustettavaksi, mutta tuli torjutuksi. Euroopasta puolestaan ostettiin pelikustantaja Virgin Mastertronic, josta tehtiin Sega Europe.

Mega Drive ehti markkinoille kaksi vuotta ennen pahinta kilpailijaansa Super Nintendoa. Vanhentuvan NES:n rinnalla Segan pelit näyttivät erityisen upeilta, ja etenkin Amerikassa tästä otettiin irti kaikki mahdollinen markkinointihyöty. *Genesis does what Nintendon't* oli osuvaa sanailua ja puri yleisöön, mutta kauppa kävi alkuun varsin vaatimattomasti: ensimmäisenä vuonna saatiin kaupaksi vain puolisen miljoonaa konetta.

Vuoden 1990 puolivälissä Sega of American johtoon nimitettiin Tom Kalinske, joka laittoi tuulemaan. Hän keräsi ympärilleen hyvän johtoryhmän, laski koneen hintaa ja käynnisti erityisesti länsimarkkinoille suunnattujen pelien tuotannon. Se naula veti, ja Segalla olikin parhaimmillaan hallussaan 65 % Pohjois-Amerikan 16-bittisten pelikoneiden markkinoista. Euroopassakin myynti veti hyvin, ja koko elinkaarensa aikana konsoli myi noin 29 miljoonaa kappaletta. Tämän myötä oli varmasti helpompi sietää se, että Japanin markkinoilla Mega Drive ei edelleenkään oikein pärjännyt.

Menestystä tuli, entäs sitten?

Voidakseen ulosmitata Mega Drivesta kaiken mahdollisen, Sega ryhtyi pian kehittelemään siihen erilaisia laajennuksia. CD-ROM oli kaikkien huulilla, joten CD-lisälaite kuulosti varmasti hyvältä ajatuk-

selta. Mega-CD (Amerikassa Sega CD) näki päivänvalon vuoden 1991 lopussa. CD-aseman lisäksi se sisälsi myös pieniä grafiikkaparannuksia, mutta suosio ei ollut mainittavaa. Suuri osa peleistä kun oli juuri niitä napin painalluksella ohjattavia suttuisia elokuvanpätkiä, joista kirjoitin Skrolli 2013.4:ssä. CD-laajennuksia myytiin alle kymmenesosa konsoleiden lukumäärästä.

Voittojen maksimoimiseksi Mega Driven tekniikkaa lisensoitiin muillekin. Ainakin Aiwa, JVC ja Pioneer tekivät omat versionsa, joissa oli erikoismausteita. Aiwan versio oli samalla radio ja CD-soitin, JVC:ssä oli karaoketoiminto ja Pioneerin Laseractive toisti Laserdisc-elokuvia. Tehtiinpä sille muutama erityinen Laserdisc-pelikin. Näiden lisäksi myös Sega itse teki samasta teemasta monia eri muunnoksia, kuten lentokoneisiin tarkoitetun Mega Jetin, mutta yksikään niistä ei saavuttanut kovin korkeita myyntimääriä.

Ristivetoa Saturnuksella

Seuraavan tuotteen ongelma on tuttu kaikille menestyneille yhtiöille, mutta harvoin tilanne pääsee kärjistymään aivan niin pitkälle kuin Segan tapauksessa. Vastakkain olivat Japanin Sega, koko yhtiön kivijalka, ja Amerikan Sega, joka oli juuri tuonut yhtiölle sen suurimman menestyksen. Aiheena taas oli se, kuinka seuraava sukupolvi otetaan haltuun.

Perinteisesti Segan tekninen osaaminen oli keskittynyt Japaniin. Siellä oli kehitetty kolmiulotteista grafiikkaa sisältäneet Virtua-sarjan kolikkopelit, jotka olivat 90-luvun alussa mullistavia. Niiden tekniikka oli kuitenkin aivan liian kallista koteihin, joten Sega oli aloittanut Hitachin kanssa projektin uuden RISC-suorittimen kehittämisestä. Saturnina tunnettuun koneeseen tulisi kaksi rinnakkaista SH-2-suoritinta sekä koko joukko muita piirejä.

Amerikassa oltiin alusta asti eri mieltä uuden arkkitehtuurin mielekkyydestä. Kalinske otti yhteyttä Silicon Graphicsiin ja kyseli apua Saturnin grafiikkapiirin tekemiseen, mutta Japanin pääkonttori torppasi sopimuksen teon. SGI lähtikin pian Nintendon kelkkaan. Tuloksena oli Nintendo 64, paljon Saturnia menestyneempi laite. Samoin puhuttiin, että kun Sony ja Nintendo riitaantuivat Playstationina tunnetun Super Nintendon CD-aseman valmistuksen tiimoilta, olisi Amerikan Sega ollut jo kärppänä neuvottelemassa siitäkin. Tokio oli kuitenkin päättänyt vakaasti, että uusi konsoli tehdään yksin. Nähtyään Playstationin prototyypin Segan johto päätti silti vielä avata Saturnin designin ja lisätä sen 3D-tehoa.

32-kertainen epäonni

Saturnin kehittely oli varsin hidasta, ja muiden valmistajien uudet konsolit kolkuttelivat jo ovella. Hätäratkaisuna päädyttiin – yllättävän yksimielisesti – Mega Driven turboahtamiseen. Konsolin päälle leivottiin 32X-nimellä tunnettu palikka, joka sisälsi muistia, kaksi SH-2-suoritinta ja muuta oheisrautaa. Tötterö tarvitsi oman virtalähteensä mutta käytti emokonsolin ohjaimia ja mahdollista CD-asemaa. Pääosin pelit kuitenkin toimitettaisiin moduuleilla. Laite oli tarkoitus saada jouluksi 1994 jenkkimarkkinoille kisaamaan 3DO:n, Jaguarin ja kumppaneiden kanssa.

32X:n kehitys jäi etupäässä Amerikan Segan vastuulle Japanin keskittyessä Saturnin viimeistelyyn. Työ oli katkonaista ja kärsi piiripulasta sekä kielimuurista. Varsinainen pommi kuitenkin putosi, kun laitetta oltiin juuri lanseeraamassa. Saturnia oltiin tuomassa markkinoille Japanissa jo 22. marraskuuta 1994, vuorokausi 32X:n Amerikan-julkaisun jälkeen. Vaikka 32X käytti osittain samoja piirejä kuin Saturn, se ei ollut ohjelmallisesti yhteensopiva, eikä Saturnissa ollut mitään porttia 32X:n pelimoduuleille.

Sega of America oli mahdottoman paikan edessä 32X:n kanssa. Ensinnäkin koko laitteesta oli tullut strategian kannalta tarpeeton. Toisekseen kuluttajat ihmettelivät, miksi heille edes markkinoidaan tällaista, kun oikea uusi konsoli oli tulossa jo muutaman kuukauden päästä myyntiin. Alle 700 000 myytyä konetta ja 40 julkaistua peliä on ehkä tämän pohjalta laskettavissa työvoitoksi, mutta reaalimaailman puolella kyse oli karvaasta ja kalliista pettymyksestä. Laajennukset valuivat hetkessä alelaareihin, ja etenkin amerikkalaiset kuluttajat alkoivat jo saada tarpeekseen Segan hyödyttömistä tuoteuutuuksista.

Peliaseman varjossa

Saturnin julkaisupäiväksi Amerikassa oli ilmoitettu toinen syyskuuta 1995, viikko ennen Playstationia. Japanin Segan hermot kuitenkin pettivät ja kone haluttiin Amerikkaan aikaisemmin, vaikka Genesis itse asiassa kävi vielä jonkin verran kaupaksi. Luotiin suunnitelma, jossa Sega sopi muutaman jakelijan kanssa 30 000 kappaleen ennakkomyynnistä E3-messujen yhteydessä toukokuussa 1995.

Segan yllätyshyökkäys meni niin pieleen kuin vain on mahdollista. Muille jälleenmyyjille ei ollut kerrottu koko asiasta, ja monet niistä suuttuivat tempauksesta Segalle ikihyviksi. Iso ketjuliike KB Toys irtisanoi myyntisopimuksensa kokonaan. Harvojen ennakkokonsoleiden ostajille oli tarjolla vain muutama Segan oma peli,

sillä edes ulkopuoliset kehittäjät eivät tienneet muuttuneista suunnitelmista. Jonoja ei siis muodostunut. Kun vielä Sony ilmoitti E3:ssa, että Playstationin julkaisuhinta on 100 dollaria Saturnia alempi, Sega oli nöyryytetty. Ostava yleisö odottaisi Sonyn halvempaa konetta. Samalla Saturnin julkaisu tappoi tehokkaasti Genesiksen jo muutenkin hiipumassa olleen myynnin.

Käytännössä peli oli tämän jälkeen selvä. Playstation tuhosi Saturnin USA:n ja Euroopan markkinoilla. Vain Japanissa Sega onnistui nousemaan minkäänlaiseksi peluriksi. Alle kymmenen miljoonan kappaleen kokonaismyynti jätti Saturnin selvästi sukupolvensa heikoimmaksi, sillä Nintendo 64 myi kolminkertaisesti, Playstation yli kymmenkertaisesti. Saturnin monimutkainen arkkitehtuuri teki pelien kääntämisestä vaikeaa, ja suositut 3D-nimikkeet olivat poikkeuksetta Playstationia kehnomman näköisiä. Pelkät Segan omat pelit eivät pitkälle riittäneet.

Sega kirjasi Saturnista satojen miljoonien tappiot, ja johtoportaassa oli hetken ajan tuulista. Tom Kalinske jätti Sega of American kokonaan, ja Japanissa leikittiin muuten vain tuolileikkiä. Alkoi olla selvää, että paukkuja olisi enää yhteen yritykseen.

Miltei viaton uhri

Saturnin seuraaja Dreamcast oli Segan kehittämistä konsoleista kenties paras. Se saattoi ylipäätään olla eräs parhaiten suunnitelluista konsoleista. Senkin kehittämiseen liittyi draamaa, kun alun perin yhteistyökumppaniksi kaavailtu 3dfx vuoti suunnitelmat julkisuuteen, mutta tilalle valittu PowerVR2 oli vuonna 1998 kaikin puolin ajantasaista rautaa sekin. Lisäksi Sega toi Dreamcastille mainion valikoiman omia pelejään. Siksi onkin raadollista, että sillä oli syntymästä asti rasitteenaan myös kaikki Segan vanhat virheet.

Yhdysvaltain julkaisu sujui kuitenkin erinomaisesti, ja Dreamcast piti pitkään hallussaan julkaisupäivänä eniten myyneen koneen titteliä. Pelkkiä ennakkovarauksia oli yli 300 000 koneesta. Euroopassakin syntyi jonkin verran kuhinaa, mutta Japani ei DC:lle lämmennyt. Rahakkaita Japanin markkinoita Sega olisi nimenomaan kaivannut.

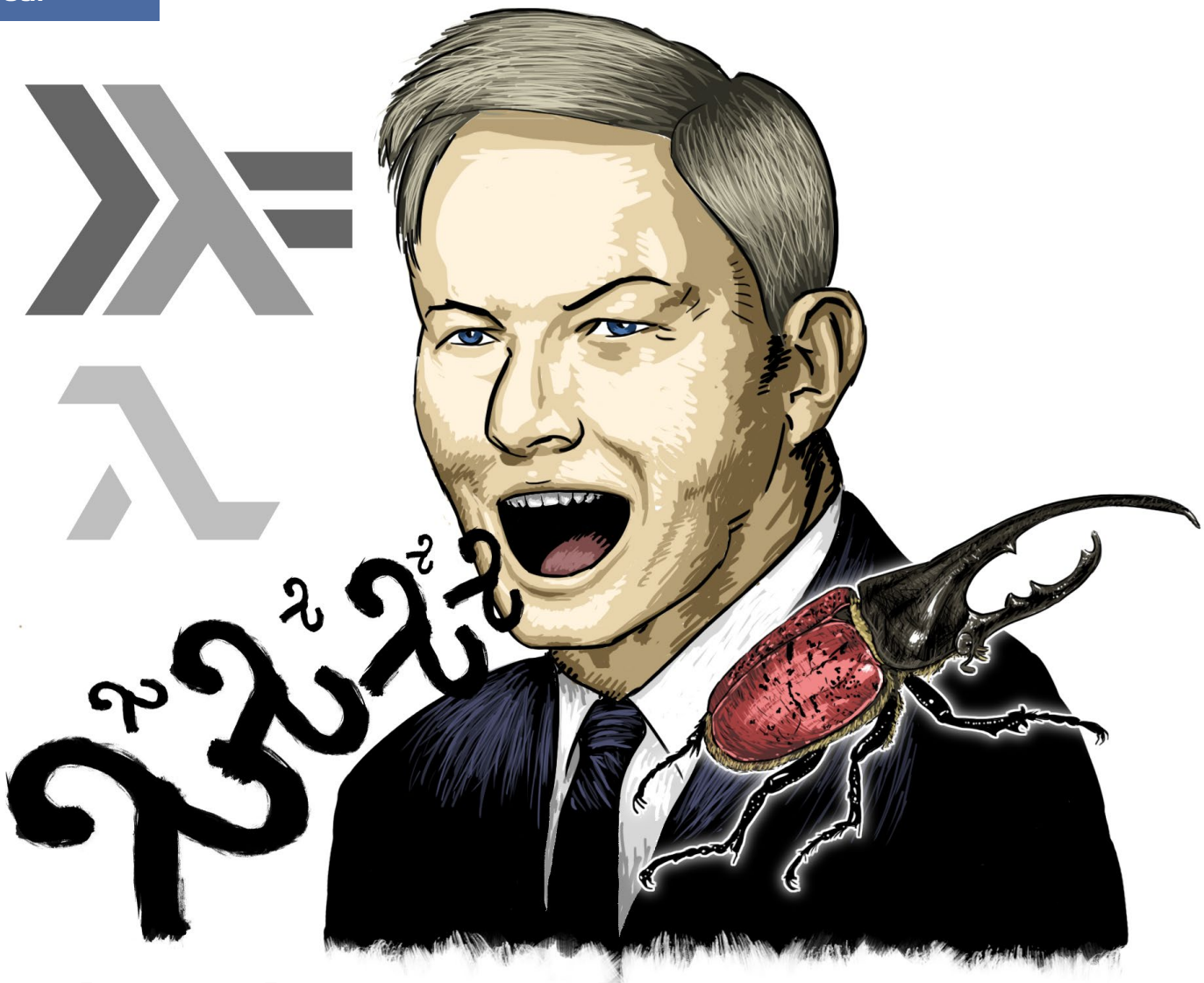
Jatkomyynti ei sujunut yhtä hyvin, sillä aiempi ”segoilu” oli karkottanut paitsi osan jälleenmyyjistä, myös Electronic Artsin. Ilman sen vuotuisia urheilupelejä moni kone jäi kauppaan, vaikka Sega yrittikin korvata niitä omilla Sega Sports -tuotteillaan. Ostavalla yleisöllä ja jälleenmyyjillä oli muutenkin kovin tuoreessa muistissa se, miten Sega-merkkiset konsolit muutamaa vuotta aiemmin happanivat hetkessä käsiin.

Lopullinen niitti oli, että Sony aloitti rummun lyömisen Playstation 2:sta. Muutamaa vuotta aiemmin Saturn oli saanut päihin Playstationilta, ja nyt pelkkä lupaus seuraajakonsolista riitti hiekoittamaan Segan ontuvan markkinakoneiston rattaat. Saturn oli kuihduttanut Segan kassan, eikä paukkuja vastata Sonyn mediahyökkäykseen ollut. Lisäksi Dreamcastista puuttui monen himoitsema DVD-asema. Kolmen peräkkäisen jättitappiovuoden ja kymmenen miljoonan Dreamcastin jälkeen Sega nosti kädet pystyyn tammikuussa 2001. DC tapettaisiin, ja yhtiö siirtyisi pelkäksi pelijulkaisijaksi.

Kunniakas loppu

Vanha Sega kaatui saappaat jalassa. Jos se ei olisi laittanut Dreamcastiin kaikkia paukkujaan, pelihistoria saattaisi kohdella sitä paljon ankarammin. Viimeisillä voimillaan se kuitenkin pusersi markkinoille sympaattisen joutsenlaulun, josta on vaikeaa olla pitämättä. Dreamcastilla syntyneet omintakeiset pelisarjat saivat jatkoa toisilla koneilla, ja Segakin pääsi viimein jälleen voitolliseksi myymällä pelejään niin Sonyn, Nintendon kuin Microsoftinkin konsoleille.

Tapahtumista kirjoittaessa on kuitenkin vaikea välttää haikeutta, sillä Dreamcast osoitti, mihin Sega pystyi silloin, kun se ei hukannut resurssejaan osastojen väliseen kissanhännänvetoon. Monen mielestä se oli viimeinen hardcore-pelikone. Saturn lieneekin siten ainoa konsoli, joka oman epäonnistumisensa lisäksi onnistui näivettämään myös teknisesti paremman seuraajansa.



Haskell – koeajo webissä

Olio-ohjelmoinnin konkari laajensi näköalojaan ja tutustui pahamaineisen teoreettiseen ohjelmointikieleen: Haskelliin. Nyt Haskell-aloittelija kertoo ensivaikutelmiaan kielestä. Kannattaako sitä opetella? Ja voiko sillä rakentaa jotain käytännöllistäkin?

Teksti: Kenneth Falck Kuvat: Risto Mäki-Petäys, Mitol Berschewsky

Kun aloitin Haskellin opiskelun, tunsin itseni samalla tavalla voimattomaksi kuin parikymmentä vuotta sitten Lispin alkeita opetellessani. Molemmat kielet tuntuivat rakenteeltaan yksinkertaisilta, mutta minun oli aluksi äärimmäisen vaikeaa ymmärtää, miten niillä luodaan omia ohjelmia.

Haskell on puhtaasti funktionaalinen ohjelmointikieli: mihinkään muuttujiin tai kenttiin ei saa sijoittaa, vaan funktioiden täytyy tuottaa samoille argumenteille aina sama paluuarvo. Rajoitusten vastapainoksi Haskellissa on kuitenkin monia hyödyllisiä ominaisuuksia, kuten hahmonsovitus, funktioiden osittaissovellus ja tiukka staattinen tyypitys. Haskell on jäänyt suosiossa yleisempien kielten varjoon, ja sillä on akateemisen, epäkäytännöllisen kielen maine.

Useimmat ohjelmoijat ymmärtävät olio-ohjelmoinnin ja proseduraalisen ohjelmoinnin periaatteet, mutta funktionaalinen ohjelmointi voi tuntua mysteeriltä. Olemme yleensä aloittaneet uramme Basicilla, C:llä, Pythonilla tai vastaavalla ohjelmointikielellä. Uuden kielen opettelu on helppoa, koska kielet rakentuvat pitkälti samojen käsitteiden varaan. Ainoastaan syntaksi on hieman erilainen.

Haskellia opetellessa on edessä kuitenkin lannistava kielimuuri. Esimerkkiohjelmien syntaksi ei tahdo avautua millään, ja niiden pienikin muokkaaminen aiheuttaa loputtomia käännösvirheitä. Toisin sanoen Haskell on aivan mahtava haaste ohjelmoijalle, joka luulee osaavansa jo kaiken!

Itse koin haasteen erityisen vaikeaksi siksi, että Haskellin ohjekirjat ovat tavalliselle ohjelmoijalle aivan vieraasta maailmasta kotoisin. Poikkeuksiakin onneksi on: paras tapa päästä alkuun on hankkia Miran Lipovačan *Learn You a Haskell for Great Good*. Se avautuu aloittelijalle helposti mutta ei käy tylsäksi kokeneellekaan koodaajalle.

Opiskelun edetessä aloin sitten oivaltaa, etteivät tavallisetkaan Haskell-ohjeet ole ihan hepreaa. Ne vain olettavat, että lukijalla on valmiiksi hallussaan kaikki Haskellin peruskäsitteet. Käsitteitä ei voi lainata minkään perinteisen ohjelmointikielen piiristä, vaan ne on opeteltava tyhjästä Haskellia varten.

Miksi web-ohjelmointi?

Haskellia syytetään usein siitä, että se sopii vain teoreettiseen ohjelmointiin. Monet Haskell-projektit varmasti ovatkin erilaisten tieteellisten algoritmien toteutuksia ja teoreettisia kokeiluja. Tällaisten ohjelmien kanssa ei välttämättä tarvitse

murehtia, pystyykö Haskellilla kytkeytymään vanhaan jäämistötietokantaan tai hyödyntämään jotain kipeästi tarvittavaa apukirjastoa.

Web-ohjelmointi puolestaan on käytännönläheistä ohjelmistotuotantoa, jossa tarvitaan vähemmän algoritmeja ja enemmän yhteistoimintaa valmiiden kirjastojen ja järjestelmien kanssa. Web-sovellus on hyvä esimerkki todellisen elämän projektista. Se voi vieläpä tarjota sovelluspalvelurajapinnan, jonka kautta muutkin ohjelmat voivat hyödyntää sen toimintoja.

Tämän vuoksi päätin itse aloittaa Haskellin opiskelun nimenomaan web-ohjelmoinnista. Halusin tietää, kuinka kieli soveltuu siihen eli mitkä tehtävät ovat helppoja ja mitkä hankalia. Toisaalta kaipasin myös uusia oivalluksia ja keinoja vanhojen tuttujen ohjelmointiongelmien ratkaisemiseen.

Yesod, sovelluksen perusta

Haskell-maailmassa on yksi varteenotettava web-sovelluskehys, Yesod. Se muistuttaa rakenteeltaan hieman Ruby on Railsia. Yesodilla voi helposti luoda toimivan esimerkkisovelluksen, johon voi sitten alkaa lisäillä omaa koodiaan. Muitakin sovelluskehyksiä on olemassa, mutta Yesod tarjoaa kerralla kaikki nykyaikaisissa web-sovelluksissa tarvittavat toiminnot.

Yesod kytkeytyy suoraan MySQL-, Postgres-, SQLite- ja MongoDB-tietokantoihin. Tämä tekee alkuun pääsemisestä helppoa, sillä kehittäjä ei joudu heti miettimään tietokannan ajurien yksityiskohtia. Riittää, että kertoo Yesodille tietokannan osoitteen ja tunnistautumistiedot sekä mallintaa halutut taulurakenteet erilliseen tiedostoon.

Yesod tarjoaa myös valmiin hakemistorakenteen web-sovelluksen näkymien määrittelyyn. Sovelluksen eri URL-osoitteiden näkymät listataan reititystiedostossa ja niitä vastaavat Handler-funktiot toteutetaan Haskell-lähdekooditiedostoissa. Handler-funktiot ovat tuttuja esimerkiksi Ruby on Railsin kontrollereista: niissä tehdään sivun näyttämiseen vaadittavat tietokantahaut ja loogiset operaatiot.

```
Category
    name Text
    title Text default=''
    UniqueCategory name
Article
    name Text
    title Text default=''
    body Text
    category CategoryId
    UniqueArticle category name
```

Mallitiedostoista tuotetaan niin SQL-skeemoja kuin Haskell-tyyppejäkin.

Perinteisissä web-sovelluksissa HTML-sivut tuotetaan yleensä tiedostoista, jotka on kirjoitettu jollakin sivupohjakielellä. CSS-tyylit ja Javascript-koodi jaetaan yleensä sellaisenaan .css- ja .js-päätteisinä tiedostoina.

Sen sijaan Yesodissa HTML, CSS ja Javascript liitetään osaksi Haskell-kääntämistä, mikä tekee sivujen tuottamisesta on kinkkisempää. Kääntäminen esimerkiksi epäonnistuu, jos pääsovelluksessa viitataan olemattomaan CSS-tiedostoon. Samanlainen virhe tulee, jos CSS-tiedostosta viitataan kuvatiedostoon, jota ei ole olemassa.

Kaikille kolmelle tiedostotyypille on Yesodissa omat sivupohjakielensä. Niiden nimissä näkyy Shakespeare-teema: Hamlet, Cassius ja Julius. Peruskirjastojen ohelle voi määritellä omia tiedostotyyppejä, esimerkiksi Coffeescript-tuen, jonka avulla Yesod osaa kääntää Coffeescript-koodin Javascriptiksi selainta varten.

Sivupohjat käännetään Haskelliksi, joten niille tehdään käännösaikaiset tyyppitarkistukset: jos niissä viitataan olemattomaan tai vääräntyyppiseen muuttujaan, saadaan käännösvirhe. Perinteisissä web-sovelluskehyksissä tällaiset ongelmat huomattaisiin vasta sitten, kun sivua yritetään luoda web-palvelun käyttäjää varten.

Haskellin vahvaa tyyppitarkistusta ei kannata pelätä liikaa, sillä sivupohjat ovat joustavia. Niihin voi sijoitella vapaasti mitä tahansa HTML-, CSS- tai Javascript-koodia sekä viittauksia tiedostoihin, jotka ladataan Yesodin ulkopuolelta.

Halutessaan voi siis rakentaa sellaisenkin sovelluksen, joka palvellaan erillisinä Javascript-tiedostoina täysin Yesodista ja Haskellista irrallaan. Käytännössä tämä onkin tärkeää, sillä yleensä web-palvelun keskeiset tiedostot halutaan sijoittaa erilliseen tehokkaaseen palvelimeen tai jakeluverkkoon.

Funktioista funktioihin

Kun olin lopulta sisäistänyt Yesodin periaatteet, sain ensimmäisen web-sovellukseni pyörimään. Muokattuani hieman sen sivupohjia tuli vihdoin aika lisätä uusia tietokantahakuja ja toimintalogiikkaa Handler-funktioihin. En voinut enää viivyttää väistämätöntä vaan jouduin viimein kirjoittamaan ihan oikeaa koodia ja

```
/static          StaticR   Static getStatic
/auth            AuthR     Auth   getAuth
/                HomeR     GET
/cat/#Text       CategoryR GET    POST
/cat/#Text/#Text ArticleR  GET    POST
```

Reitityskuvaus varmistaa, että jokaisella sivulla on oikeantyyppinen käsittelijä.

tutustumaan Haskellin pimeään puoleen: funktionaalisen ohjelmoinnin outouksiin.

Eräs outouksista on se, että Haskellissa funktioilla on vain yksi parametri ja yksi paluuarvo. Tämä jää yksinkertaisessa web-ohjelmoinnissa helposti huomaamatta, sillä funktioita voi luontevasti käyttää niin kuin ne ottaisivat vastaan useampia parametreja. Tämä on kuitenkin Haskellin opiskelussa niin tärkeä asia, että se kannattaa sisäistää heti alussa.

Hyvä esimerkki usean parametrin funktiosta voisi olla vaikkapa web-sovelluksen Handler-funktio, joka saa argumentteina sisältökategorian nimen sekä haetun artikkelin tunnisteen ja palauttaa artikkelisivun:

```
getArticleInCategory :: Text -> Text -> Widget
getArticleInCategory catName articleName = ...
```

Funktiota voi kutsua kahdella argumentilla, jolloin se aivan odotetusti palauttaa halutun sivun:

```
haskellPage = getArticleInCategory
              "programming" "haskell-tutorial"
```

Oikeasti kyseessä on kuitenkin yksipaikkainen funktio, joka palauttaa toisen yksipaikkaisen funktion. Sitä voi siis kutsua myös yhdellä argumentilla:

```
getProgrammingArticle :: Text -> Widget
getProgrammingArticle =
  getArticleInCategory "programming"
```

Tällöin paluuarvona ei olekaan mikään artikkelisivu vaan ”osittaissovellettu” funktio, joka hakee artikkelin kiinteästä kategoriasta. Osittaissovellettua funktiota voidaan kutsua jäljellä olevalla argumentilla, jolloin saadaan sama lopputulos kuin alkuperäisestä kutsusta. Kutsuja voidaan sitten tehdä useampiakin ja eri argumentein:

```
haskellTutorial =
  getProgrammingArticle "haskell-tutorial"
erlangTutorial =
  getProgrammingArticle "erlang-tutorial"
cTutorial =
  getProgrammingArticle "c-tutorial"
```

Kun idean on omaksunut, saa osittaissovelluksella usein luotua näppäriä apufunktioita kuin itsestään.

Mystiset monadit

Oudot funktiokutsut ovat kuitenkin lastenleikkiä verrattuna Haskellin kuuluisimpaan omituisuuteen: monadeihin. Monadi on Haskellin kaikkein vaikein käsite ja samalla myös vaikeimmin selitettävä. Tunnettu Javascript-guru Douglas Crockford onkin todennut, että kun ohjelmoija kerran ymmärtää monadit, hän menettää samalla kyvyn selittää niitä muille. Monadeja kuitenkin käytetään Yesodissa kaikkialla, joten yritettävä on.

Monadi on tapa paketoida arvoja niin, että varsinaiseen arvoon liittyy myös jonkinlainen lisätieto tai asiayhteys. Esimerkiksi Maybe-monadi luo paketin, jossa voi olla arvo tai se voi olla tyhjä. Lähin oliomaailman vertauskuva olisi olioviite, joka voi osoittaa olioon tai joka voi olla *null*.

Monadien ideana on, että ohjelmakoodi voi joko olla kiinnostunut lisätystä kontekstista tai se voi olla kokonaan välittämättä siitä. Haskell tarjoaa ohjelmoijalle erilaisia työkaluja, joiden ansiosta funktiokutsujen ei tarvitse välttämättä välittää muuttujien monadikuorrutteesta, vaan funktion suorittama operaatio voidaan kohdistaa monadin sisällä olevaan arvoon.

Learn You a Haskell -kirja käy monadien idean läpi perusteellisesti ja ymmärrettävästi. Se kuitenkin käyttää esimerkkeinä vain melko yksinkertaisia monadeja. Valitettavasti Yesodin monadityypit ovat kuitenkin monimutkaisempia ja niiden ymmärtäminen on vaikeampaa.

Onneksi monadeja voi kuitenkin käyttää, vaikka niitä ei täysin ymmärtäisikään. Yesodin Handler-funktiot perustuvat monadeihin, mutta Haskellin *do*-notaation ansiosta niitä voi käyttää melko helposti. *do*-notaatio on vaihtoehtoinen syntaksi, jolla Haskell-koodin saa näyttämään perinteiseltä imperatiiviselta koodilta, jota suoritetaan lause kerrallaan. Se on kuitenkin pelkkää harhaa: konepellin alla lauseet muutetaan kryptiseksi sarjaksi sisäkkäisiä funktiokutsuja.

```
$if null articles
  <p> There are no articles in this category.
$else
  <ul>
    $forall art <- articles
      <li>
        <a href="@{ArticleR (categoryName cat) (articleName art)}">
          #{articleTitle art}
```

Hamlet-sivupohjissa sivun rakenne ilmaistaan sisennystasoilla.

```
articleForm :: CategoryId -> Html ->
               MForm Handler (FormResult Article, Widget)
articleForm catId = renderDivs $ Article
  <$> areq textField "Name" Nothing
  <*> areq textField "Title" Nothing
  <*> (unTextarea <$> areq textareaField "Body" Nothing)
  <*> pure catId
```

Myös lomakkeet määritellään tyyppiturvallisesti.

Haskellin hankaluuksia

Näin aloittelijana on todella vaikeaa lukea muiden kirjoittamaa Haskell-koodia. Suurin ongelma syntyy siitä, että funktioiden argumenttien ja paluuarvojen tyypit ovat usein yleisluontoisia. Sama funktio voi siis ottaa vastaan monia erityyppisiä arvoja, ja se voi myöskin palauttaa mo-

```
postCategoryR :: Text -> Handler Html
postCategoryR catName = do
  Entity catId category <- runDB $ getBy404 (UniqueCategory catName)
  ((FormSuccess newArticle, _), _) <- runFormPost $ articleForm catId
  _ <- runDB $ insert newArticle
  redirect $ ArticleR (categoryName category) (articleName newArticle)
```

Tietokantaan syöttäminen onnistuu funktionaalisestikin.

```
getBy404
  :: (MonadTrans t, PersistUnique (t (HandlerT site IO)),
      PersistEntity val,
      PersistMonadBackend (t (HandlerT site IO))
      ~ PersistEntityBackend val) =>
     Unique val -> t (HandlerT site IO) (Entity val)
```

Apufunktioita on helppo käyttää, mutta niiden tyyppejä ei kuolevainen ymmärrä.

nia erityyppisiä arvoja. Tyyppijärjestelmä ei ole samalla tavalla hierarkkinen kuin oliokielissä, joten sitä ei voi ajatella puurakenteena. Tyypit ovat enemmänkin kuin Lego-palikoita, joita voi liittää yhteen monilla eri tavoilla.

Haskellissa on lisäksi ikäviä ominaisuuksia, jotka tekevät lähdekoodin lukemisesta entistäkin vaikeampaa. Pahinta on se, että ohjelman nimiavaruuteen voi tuoda monien kirjastojen kaikki funktiot. Kun ohjelmakoodissa sitten esiintyy vaikkapa nimi *Object* tai *run*, joutuu käymään läpi parikymmentä kirjastoa selvittääkseen, mistä ne oikein tulevat.

Haskellin nimiavaruuksien alkeellisuus johtaa myös siihen, että tietokantaan tallennettuja tietorakenteita on kömpelöä määritellä. Käytännössä jokaisella tietokantakentällä on oltava nimi, joka yksilöi sen globaalisti. Kentän nimi ei siis voi olla monessa eri paikassa title, vaan sen täytyy olla artikkeleissa *articleTitle*, kommenteissa *commentTitle*, kategorioissa *categoryTitle* ja niin edelleen.

Tietorakenteiden käyttö on muutenkin hankalaa oliokieleen tottuneelle, koska Haskellissa ei ole omaa syntaksia tietue- tai taulukkoviittauksille vaan kaikkea käsitellään funktiokutsuilla. Esimerkiksi muodon

```
article.comments[0].title
```

sijaan täytyykin kirjoittaa

```
commentTitle (head (articleComments article))
```

Onneksi tähän kuitenkin tottuu hiljalleen, eivätkä kutsut sitten enää näytä yhtä kummallisilta kuin aluksi.

Kehityksen kääntöpuolia

Haskell on elävä ohjelmointikieli, ja siihen kehitetään koko ajan uusia laajennoksia ja kirjastoja. Kehittyminen on hyväksi, mutta valitettavasti se myös tekee Haskellin opiskelusta monimutkaisempaa. On tyypillistä, että Haskell-kääntäjästä pitää kytkeä päälle kymmenen eri laajennosta, jotta ohjelma suostuu kääntymään. Laajennokset tekevät kielestä monipuolisemman mutta myös vaikeammin ymmärrettävän.

Esimerkiksi Yesod käyttää kahta erilaista Haskell-lähdekoodin esikäsittelyvaihetta. Template Haskell -makrojärjestelmää tarvitaan monessa paikassa, mutta toisaalta koodissa on myös *#ifdef*-direktiivejä, jotka ajetaan perinteisen C-esikääntäjän läpi. Kieltä opiskellessa täytyy pitää varansa, ettei sekoita Yesodin erikoissyntakseja tavalliseen Haskell-koodiin.

Usein ohjelmat tarvitsevat laajennosten lisäksi myös kymmenittäin apukirjastoja. Kirjastot saattavat olla keskenään yhteensopivia vain tiettyjen versioiden osalta, joten yhdenkin kirjaston päivittäminen voi rikkoa koko sovelluksen. Saattaa siis käydä niin, että yhtenä päivänä *cabal install* -komento asentaa kirjastoja ongelmitta ja seuraavana se ei enää toimi ollenkaan. Turvallisinta onkin eristää Haskell-sovellukset toisistaan niin, että kullakin sovelluksella on omat versionsa kirjastoista.

Järkeä vai ei?

Muutaman kuukauden mittaisen tutustumisen jälkeen en pidä Haskellia mitenkään ihanteellisena ohjelmointikielenä web-sovellusten kehittämiseen. Kielellä on äärimmäisen jyrkkä oppimiskäyrä, sen apukirjastot ovat rajoittuneita ja sillä kehitettyjä sovelluksia on vaikeaa päivittää.

Vaikeuksien vastapainona on kuitenkin ohjelmakoodin tehokkuus ja turvallisuus. Haskellin äärimmäisen tiukka tyyppitarkistus löytää monenlaiset ajatusvirheet ennen kuin ohjelmaa tarvitsee edes käynnistää. Ohjelmointi saattaa olla työläämpää kuin dynaamisemmilla kielillä, mutta kun ohjelman on kerran saanut kääntymään, suorituksen aikaisia virheitä ilmenee vain harvoin.

Nykyaikaiseen nopeatahtiseen web-kehitykseen Haskell sopii kuitenkin kehnosti. Nopea prototyyppikoodin testailu takkuilee, koska joutuu odottelemaan koodin kääntymistä jokaisen muutoksen välissä. Pienet muutokset ovat työläitä, sillä ohjelmoija joutuu kirjoittamaan jokaisen uuden funktion nimen kahteen kertaan ja lisäämään uudet moduulit sekä lähdekoodiin että pakettikuvaustiedostoihin. Tällä tavoin ei mitenkään päästä esimerkiksi Ruby on Railsin käyttömukavuuteen tai Flaskin yksinkertaisuuteen.

Nähdäkseni Haskellin tulevaisuus web-kehityksessä liittyy sellaisiin projekteihin, joissa tarvitaan luotettavia ja erikoistuneita taustajärjestelmiä. Esimerkiksi Mochiweb-mainosverkosto ja CouchDB-tietokanta on kumpikin toteutettu Erlang-kielellä, joka muistuttaa monessa suhteessa Haskellia. Molemmat tarjoavat julkiseen verkkoon vain sovelluspalvelurajapinnan ja pyörittävät sen takana tehokasta taustajärjestelmäänsä.

Linkkejä

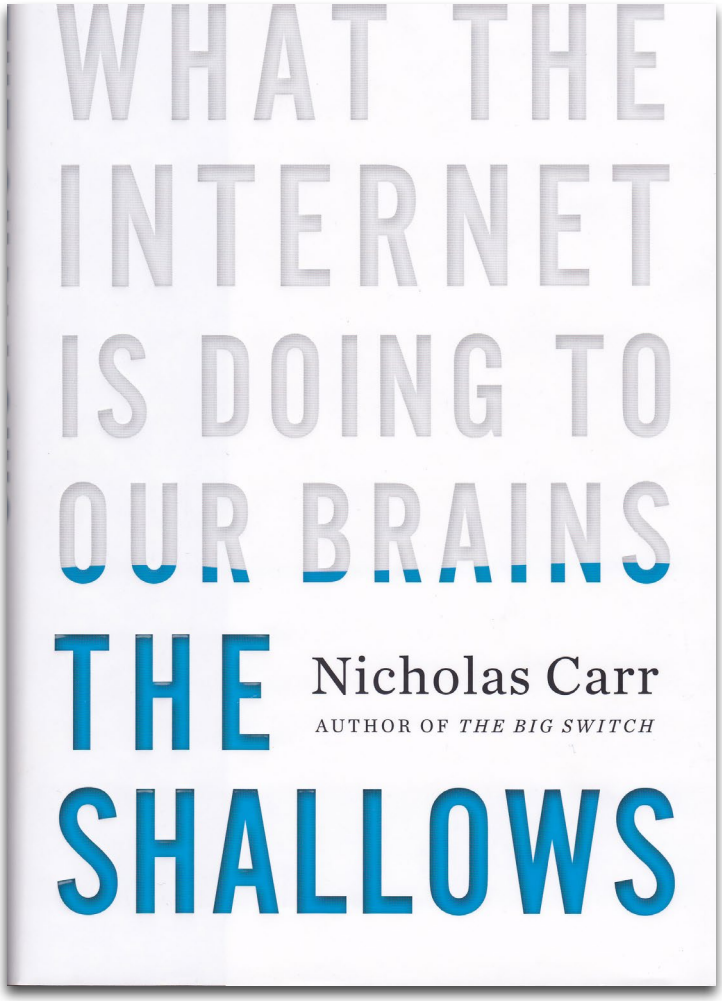
- Haskell: http://haskell.org/
- Learn You a Haskell for Great Good: http://learnyouahaskell.com/
- Yesod: http://www.yesodweb.com/

Omissa harrastusprojekteissa on kuitenkin mukavaa kokeilla Haskellia tavallistenkin sivustojen toteuttamiseen, ja Yesod tekee siitä kohtalaisen yksinkertaista. Palkintona vaivannäöstä saa vähintäänkin aivan uudenlaisen näkökulman ohjelmointiin.

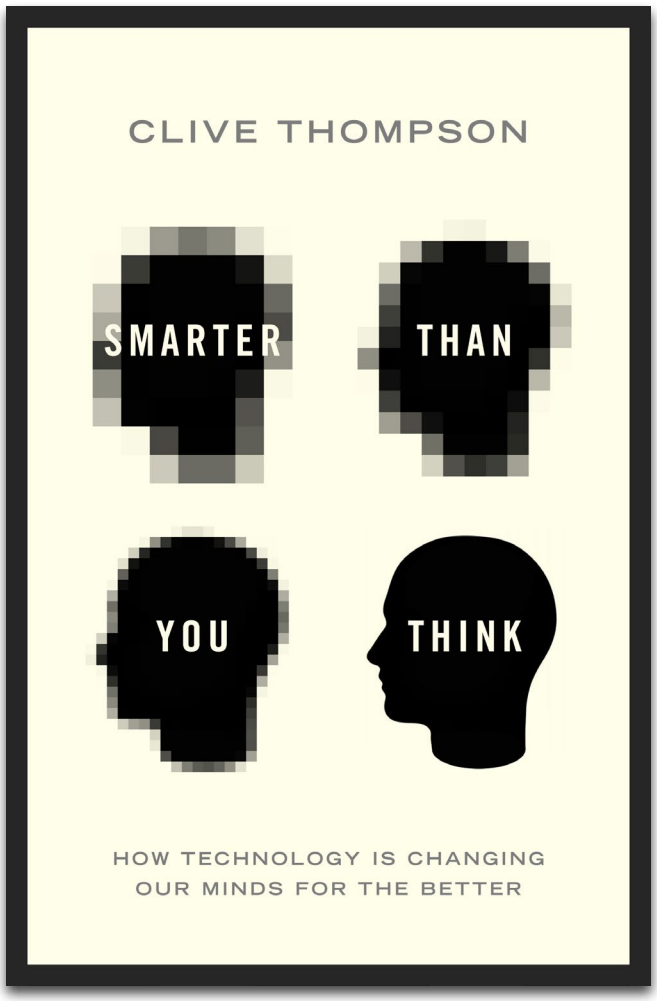
Tietotekniikka muuttaa meitä ja maailmaa

Internetin jatkuva läsnäolo huonontaa muistia ja laiskistaa oppijaa! Tai ehkä ihmisen ja koneen lähentyminen avaakin ihan uusia viestinnän ja osaamisen tapoja?

Teksti: Jussi Määttä



Nicholas Carr: **The Shallows: What the Internet is Doing to Our Brains** (W. W. Norton, 2010)



Clive Thompson: **Smarter Than You Think: How Technology is Changing Our Minds for the Better** (Penguin, 2013)

Teknologia ja aivot

Aivot muuttuvat jatkuvasti. Uusia yhteyksiä syntyy tarpeen mukaan, ja vähälle käytölle jääneet osat delegoituvat muihin tehtäviin. Ilmiötä kutsutaan neuroplastisuudeksi. Toisin kuin aiemmin luultiin, aivot jatkavat sopeutumistaan myös aikuisiällä.

Vanha vitsi kuuluu, että vasara kädessä kaikki ongelmat näyttävät nauloilta. Totta onkin, että mikä tahansa käteen otettu työkalu muuttuu aivojen näkökulmasta käden osaksi. Jokainen työkalu tuo uusia mahdollisuuksia (vasaralla voi iskeä nauloja), mutta toisaalta myös rajoituksia (neulominen vaikeutuu huomattavasti). Ja aivot oppivat nopeasti. Kokeile vaikka pitää vasaraa kädessäsi pari tuntia. Uuteen asetelmaan tottuu pian ja arki sujuu, mitä nyt ulkopuoliset saattavat oudoksua.

Jos tietokirjailija Nicholas Carrilta kysytään, olemme tottuneet internetiin liian hyvin. Kirjassaan *The Shallows* (2010) hän esittää, että netti heikentää keskittymiskykyämme, huonontaa muistiamme ja aiheuttaa tarpeetonta rasitusta mielellemme.

Optimistisempaa koulukuntaa edustaa Clive Thompsonin kirja *Smarter Than You Think* (2013). Thompsonin mielestä uuden teknologian hyödyt ovat moninkertaisesti haittoja suuremmat. Hänelle ihmisen ja koneen yhdistelmä on parempi kuin kumpikaan yksinään, ja netin viestintämahdollisuudet tuovat valtavia parannuksia monille yksityisen ja julkisen elämän alueille.

Ihmisen ja koneen hybridi

IBM:n Deep Blue -shakkitietokone selätti Garri Kasparovin vuonna 1997. Ihmisen tappio koneesta sai monissa aikaan tappiomielialan. Tarina ei kuitenkaan pääty tähän, kuten Thompson kertoo kirjassaan. Kasparov liittoutui vihollisensa kanssa: kone laskekoon vaihtoehtoisia siirtoja, ihmisen intuitio valitkoon niistä

viisaimman. Allianssi osoittautui voitokkaaksi niin ihmisiä kuin koneitakin vastaan.

Toinen näkökulma löytyy eräästä Carrin kuvaamasta tutkimuksesta, jossa koehenkilöt ratkoivat vaikeaa logiikkapähkinää tietokoneella. Osa heistä sai koneelta apua: esimerkiksi lailliset siirrot osoitettiin visuaalisesti. Toinen ryhmä käytti riisuttua ohjelmaversiota ilman helpotuksia. Aluksi jälkimmäinen ryhmä pärjäsi huonommin, mutta harjoituksen myötä pähkinät alkoivat ratketa ”apuratasryhmää” nopeammin ja vähemmin virhein. Mitä fiksumpi ohjelma, sitä tyhmempi käyttäjä.

Muisto vain

Unohtuneen faktan salamannopea tarkistaminen netistä on arkisempi esimerkki aivojen ”laajentamisesta” tekniikan avulla. Thompson on innoissaan tällaisista keinoista kompensoida aivojen ailahtelua. Tietokoneet eivät muista väärin, eikä mitään tarvitse poistaa, kun tallennuskapasiteettia on riittämiin. Mutta kuten Thompson huomauttaa, datan taltiointi ei itsestään johda enempään muistamiseen. Ehkä kuitenkin riittää muistaa, mitä tietoa on olemassa ja mistä se löytyy. Jospa tietokoneet voisivat auttaa tässäkin?

Carr puolestaan varoittaa vähättelemästä perinteisen muistamisen merkitystä. Aivoissa kaikki tiedonmuruset kytkeytyvät toisiinsa ja järjestyvät jatkuvasti uudelleen synnyttäen ”ahaa-elämyksiä”. Tämä kuitenkin edellyttää, että tieto maltetaan ensin siirtää pitkän aikavälin muistiin. Jatkuva googletus johtaa muistin heikkenemiseen, mikä taas lisää googletuksen tarvetta. Noidankehä on valmis.

Suhtautuminen muistiin on muuttunut aikojen saatossa. Carr siteeraa tunnettua 1800-luvun ajattelijaa: ”Muistamisen taito on ajattelemisen taito.” Aikanaan lausahdus oli itsestäänselvyys. Nyt se kuulostaa vanhanaikaiselta.

Hyperteksti häiritsee

Netti ei olisi netti ilman linkkejä. Wikipediaa lukiessa tulee usein avattua kymmeniä sivuja uusiin välilehtiin. Googlen alkuperäisen PageRank-algoritmin menestys perustui linkkien analysointiin. Ei kai hypertekstistä mitään haittaa voi olla?

Kyllä voi, sanoo Carr ja pamauttaa tiskiin pinon tutkimuksia. Luetun ymmärtäminen heikkenee linkkien määrän kasvaessa. Jos teksti jaetaan useisiin toisiinsa linkitettyihin sivuihin, koehenkilöt ymmärtävän sisällön huonommin ja pitävät tekstiä sekavampana. Hyperteksti pakottaa lukijan tekemään jatkuvasti päätöksiä: klikatako vai ei? Linkkien erilainen ulkoasu häiritsee keskittymistä.

Toisaalta Carr myöntää, että nettitekstiä lukemaan tottuneet löytävät pitkästä tekstistä nopeasti etsimänsä. He pystyvät päättelemään hetkessä, miten todennäköisesti haluttu tieto löytyy tietyltä sivulta. Tällä tavoin löydetty informaatio kuitenkin harvemmin tallentuu pitkän aikavälin muistiin.

Thompsonkin myöntää rauhallisen, lineaarisen lukemisen merkityksen syvälliselle oppimiselle. Välillä tulisi pitää taukoja informaatiotulvalle altistumisesta ja keskittyä luontoon, kirjoihin ja taiteeseen. Ei puroakaan yritä kukaan kuiviin juoda, Thompson vertaa.

Kansankynttilöiden korvaajat

Koulujen ja tietotekniikan pyhästä liitosta on Suomessakin puhuttu jo 1990-luvulta alkaen. Investointien todellisesta hyödystä voi olla montaa mieltä, mutta nyt laajamittainen paradigman muutos saattaa olla lähempänä kuin koskaan. Thompson kuvaa kirjassaan, kuinka monissa kouluissa perinteinen asetelma on käännetty päälaelleen Khan Academy -verkkosivun avulla. Matematiikan opetusvideot katsotaan kotona ja tehtävät tehdään tunneilla. Opettajalla on enemmän aikaa yksilölliseen neuvomiseen. Oppimistulokset näyttävät paranevan, ja oppilaiden motivaatio kohenee. Tavanomaisilla oppitunneilla etenemistahti on osalle aina liian nopea tai hidas.

> *Nettitekstiä lukemaan tottuneet pystyvät päättelemään hetkessä, miten todennäköisesti haluttu tieto löytyy tietyltä sivulta.*

Matematiikka on kiitollinen oppiaine tietokoneistamisen kannalta, mutta äidinkieli on paljon haastavampi kohde. Yrityksiä on silti ollut. Surkuhupaisa esimerkkinä Carr mainitsee brittiläisen Edexcelin vuonna 2009 lanseeraaman automaattisen ainekirjoitusten arvostelijan. Sen tarkkuus oli väitetysti yhtä hyvä kuin ihmistarkastajien. Keskimäärin ohjelma saattaa toimia hyvinkin, mutta päättömyyksiä tai hienoja nyansseja se tuskin tunnistaa kenenkään kirjoitelmista.

Maailma muuttuu bitti kerrallaan

Egyptin vuoden 2011 vallankumousta ei ole suotta kutsuttu Facebook-vallankumoukseksi. Thompson kertaa kirjassaan ansiokkaasti tapahtumien tätä aspektia. Yksittäisen ihmisen on vaikea ryhtyä vastustamaan vahvaa vallanpitäjää. Facebook-ryhmän kautta egyptiläiset huomasivat silti, etteivät olleet yksin tyytymättömyydessään Mubarakin hallintoa kohtaan.

Thompson huomauttaa myös, että tiedon levittämisellä Youtuben ja Twitterin kaltaisissa palveluissa on eräs merkittävä etu. Niiden käytön estäminen voi vetää barrikadeille aktivistien lisäksi raivostuneita kissavideoiden ystäviä.

Egyptissä kansa meni kaduille Facebookin kautta. Mutta onko tyypillisempi nettiaktivismi pikemminkin passivismia, jossa pyritään parantamaan maailmaa peukkunappia painamalla?

Ei välttämättä. Vaikka statuspäivitykset ovat usein yksinään tarkasteltuina turhia, niitä pidemmän aikaa seuraamalla oppii Thompsonin mukaan tuntemaan toisen tapoja ja ajatusmaailmaa aivan uudella tavalla. Hän kutsuu ilmiötä ihmisten lisääntyneeksi tietoisuudeksi ympäristöstään (engl. ambient awareness). Sen seurauksena ihmisten harhaluulot vaikkapa puolituttujen suhtautumisesta vallanpitäjiin karsiutuvat ja kynnys puhua asioista avoimesti laskee.

Kokonaiskuva

Carrin teoksen *The Shallows* perimmäinen viesti on, että meidän täytyy oppia näkemään teknologian vaikutukset mieliimme ja toimintaamme. Carr itse huomasi netinkäytön verottaneen keskittymiskykyään ja vetäytyi Kalliovuorille pätkivän nettiyhteyden ääreen saadakseen kirjansa valmiiksi. Hän kutsuukin lukijan mukaan kapinaan hyperaktiivisuutta vastaan. Kirjalla kestää tovi päästä asiaan, ja tekijä tuntuu kohtuuttoman leipääntyneeltä erityisesti Googleen, mutta kokonaisuus on toimiva ja viesti varmasti tärkeä.

Thompsonin *Smarter Than You Think* on myös mukiinmenevä kirja, joka pyrkii näkemään asiat positiivisesti vastapainona Carrin kaltaisille teknologiapessimisteille. Selkeitä vasta-argumentteja ei Thompson juurikaan onnistu esittämään, vaikka opus vaikuttaa tehdyn juuri *The Shallowsin* haastajaksi. Thompson lähinnä kannustaa optimismiin: maailma on muuttunut ennenkin ja lopputulema on aina ollut hyvä. Hieman hämmentävästi hän päätyy vihjailemaan rivien välistä, että Carrin pessimismi on vain yritys näyttää älykkäältä. Kaiken kaikkiaan *Smarter Than You Think* on kuitenkin suositeltavaa lukemistoa aiheesta kiinnostuneille.

Cooler Master - Uncompromising Ergonomics

MIZAR

CM Storm Mizar
Laser pelihiiri

49,90 €

Mizar ammentaa inspiraationsa klassisesta oikeakätisestä ja ergonomisesta tyylistä. Tarkka laser sensori ja säädettävä DPI takaavat varmat osumat!

Ominaisuudet:

Avago 9800 Laser sensori, 7 ohjelmoitavaa näppäintä, DPI 200 - 8200, max. seurantanopeus: 150 ips / 30g

ALCOR

CM Storm Alcor
Optinen pelihiiri

37,90 €

Alcor sopii pelaamiseen erinomaisesti kätesi koosta tai grip-tyylistäsi riippumatta. Sen ergonomisen muotoilun ansiosta saat mukavan otteen hiirestä!

Ominaisuudet:

Avago 3090 Optinen sensori, DPI 400 - 4000, max. seurantanopeus 60 ips / 20g

Jimm's Trance Gamer

Jimm's Gamer -sarjan koneet ovat olleet jo pitkään pelaajien suosiossa ja suunnannäyttäjinä. Gamer -koneet ovat niittäneet mainetta arvosteluissa ja ovat moninkertaisia **lehdistön -testivoittajia!**

Trance Gamerista poistettiin vanhanaikainen savikiekko ja järjestelmälevyksi asennettiin nopea SSD-asema!

Ominaisuudet:

Intel i7-4770K 3.5GHz, -prosessori | Intel Z87-piirisarjan emolevy | NVIDIA GTX 770 2Gt -näytönohjain | 120Gt, 2.5", Sata III -SSD asema | 8Gt (2x4Gt) DDR3 1600MHz, CL9 -keskusmuisti | Cooler Master Silencio 550 **Jimm's Edition** -kotelo | Windows 7 Home Premium 64-bit (Suomi)

1259 €

JIMM'S
www.jimms.fi

Asiakaspalvelu

+358 29 70 70700
asiakaspalvelu@jimms.fi

Jimm'siltä löytyy nettitilauksiin aina toimituskuluton vaihtoehto!

Noutopalvelumyymälä Turku

Lukkosepänkatu 7
20320 Turku

Hinnat €/kpl, sis. alv24%